ON THE COMPUTATIONAL COMPLEXITY OF SCHEME EQUIVALENCE

Robert L. Constable
Cornell University

Harry B. Hunt, III
Princeton University

Sartaj Sahni
University of Minnesota

TR 74-201

March 1974

Department of Computer Science
Cornell University
Ithaca, New York 14850

ON THE COMPUTATIONAL COMPLEXITY

OF SCHEME EQUIVALENCE

Robert L. Constable
Cornell University

Harry B. Hunt, III
Princeton University

Sartaj Sahni
University of Minnesota

Abstract:

We consider the computational complexity of several decidable problems about program schemes and simple programming languages. In particular we show that the equivalence problem for Ianov schemes is NP-complete, but that the equivalence problem for strongly free schemes, which approximate the class of Ianov schemes which would actually be written, can be solved in time quadratic in the size of the scheme.

We also show that many other simple scheme classes or simple restricted programming languages have polynomially complete equivalence problems. Some are complete for the same reason that Ianov schemes are complete and some are complete for other reasons.

This research was partially supported by NSF grant GJ-579 and by an NSF graduate fellowship in Computer Science at Cornell University.

§1 Introduction

Early work with program schemes was motivated by a quest for optimization techniques, see [3,6,8]. Ideally one would find a class of schemes rich enough to include many ordinary programs yet simple enough to have a decidable equivalence problem. No attempt was made to assess the difficulty of the known decidable problems (such as equivalence for Ianov schemes [8,9], for free monadic recursion schemes ([1], etc.). However since the work of Cook [2], it has become possible to talk about matters of feasibility and tractability in a theoretically meaningful way. In this paper we consider the complexity of several decidable problems about program schemes and simple programming languages.

Our work has revealed several interesting findings. In Section 2 we show that the equivalence problem for Ianov schemes, though decidable, is NP-complete. This is true even for schemes without loops. Thus the reason why this problem is hard is quite different than the reason why equivalence of multi-variable schemes is undecidable. We prove a metatheorem that yields sufficient but general conditions on a predicate P on the Ianov schemes for P to be NP-hard. The predicates satisfying this theorem include halting, divergence, equivalence to the identity scheme, etc. An analogous theorem yields sufficient conditions on a predicate P on the multi-variable program schemes to be undecidable. Also an NP-complete variant of the Post's Correspondence Problem is introduced and used to show that freedom for 2 variable loop-free schemes is NP-complete.

In Section 3 we isolate subclasses of Ianov schemes for which the equivalence problem can be solved in polynomial time. We suspect that the "naturally occurring Ianov schemes" have this property. Our candidate for natural schemes are the <u>free Ianov schemes</u>, i.e. those in which no predicate is tested twice on the same value (in a free interpretation). But we are only able to show that strongly free Ianov schemes (those with function evaluations between tree like clusters of predicates) have a polynomial time equivalence test. To show this we carefully consider the relationship between Ianov schemes and finite automata.

In Section 4 sufficient conditions for a class of simple programming languages to have an NP-hard equivalence problem are presented. One class of programs satisfying these conditions is the LOOP1 class of Tsichritzis [12].

We first state several definitions needed in the rest of the paper.

<u>Definition 1.1:</u> <u>P</u>(<u>NP</u>) is the class of all languages over $\{0,1\}$ accepted by some deterministic (nondeterministic) polynomially time-bounded Turing machine.

<u>Definition 1.2:</u> Let Σ,Δ denote finite alphabets. Let $\Pi_{\Sigma,\Delta}$ be the class of all functions from Σ^* into Δ^* computable by some deterministic polynomially time bounded Turing machine. We say L_1 is <u>p-reducible</u> to L_0 (written $L_1 \underset{\text{ptime}}{\leq} L_0$) if there exists f in $\Pi_{\Sigma,\Delta}$ such that $x \in L_1$ iff $f(x) \in L_0$. A language L_0 is <u>NP-hard</u> if $NP \underset{\text{ptime}}{\leq} L_0$ (i.e. $\forall\ L \in NP\ \ L \underset{\text{ptime}}{\leq} L_0$). L_0 is <u>NP-complete</u> if L_0 is both NP-hard and is accepted by some nondeterministic polynomially time bounded Turing machine.

§2 Program and Recursion Schemes

We classify the complexity of several different decidable predicates on the tree, free, single and multiple variable flow chart schemes. We assume the reader is familiar with program schemes, interpretations, and the standard results concerning Herbrand or free interpretations as presented in [1], [5], or [6]. Program schemes are finite sequences of

(i) assignment instructions

$$k.\quad y \leftarrow f(x_1,\dots,x_n)\ ,$$

where k is a numeral, x_i and y are individual variables, and f is a function variable;

(ii) conditional instructions

k. IF $P(x_1,\dots,x_n)$ THEN GO TO k_1
ELSE GO TO k_2,

where k, k_1, k_2 are numerals, P is a predicate variable, and the x_i's are individual variables, and

(iii) <u>halt instructions</u>

k. LOOP,

where k is a numeral. Instructions of the form k. LOOP are considered as abbreviations for instructions of the form

k. IF $P(x_1,\ldots,x_n)$ THEN GO TO k ELSE GO TO k.

Finally, we frequently assume that the label of the <u>ith</u> element of such a sequence is the binary numeral for i, the first element of the sequence is the unique start or initial statement, and the last element of the sequence is either a loop or halt instruction.

<u>Definition 2.1:</u> A <u>tree scheme</u> is a program scheme such that
(i) for each statement, there is exactly one way it can be reached from the start statement and
(ii) the last statement in each maximal path is either a halt or a loop statement.

The semantics for our program schemes is the conventional one. Thus, an interpretation I of a program scheme S consists of
(a) a nonempty set of elements D_I together with
(b) assignments of fixed elements of D to the individual variables of S, of functions $f_I\ D^n \to D$ to the function variables f of S, and of functions from D^n into $\{T,F\}$ to the predicate variables of S. The pair $P = \langle S,I\rangle$, where S is a program scheme and I is an interpretation of S, is called a <u>program</u>. Let the number of distinct individual variables in S be n. Given $\sigma \in I^n$ for the individual variables of S, the program is executable. Thus we also talk about computations $\langle S,I,\sigma\rangle$. The value of a computation, denoted by

$\langle S,I,\sigma\rangle$, is the n-tuple of final values of the individual variables $X_1,\ldots,X_n$ of S. If the computation does not terminate then Value $\langle S,I,\sigma\rangle$ is undefined.

Definition 2.2: For given programs $\langle S,I\rangle$ and $\langle S',I'\rangle$ we say that

(i) $\langle S,I\rangle$ halts if for every assignment of initial values $\sigma \in D^n$ to the variables of S Value $\langle S,I,\sigma\rangle$ is defined;

(ii) $\langle S,I\rangle$ diverges if for every assignment of initall values $\sigma \in D^n$ to the variables of S Value $\langle S,I,\sigma\rangle$ is undefined;

(iii) $\langle S,I\rangle$ and $\langle S',I'\rangle$ are (strongly) equivalent if for every assignment of initial values $\sigma \in D^n$ to the variables of S,S' Value $\langle S,I,\sigma\rangle \cong$ Value $\langle S',I',\sigma\rangle$ † ; and

(iv) $\langle S,I\rangle$ and $\langle S',I\rangle$ are isomorphic if for every assignment of initial values $\sigma \in S^n$ to the variables of S, the sequences of instructions executed in the finite or infinite computation of $\langle S,I,\sigma\rangle$ and $\langle S',I,\sigma\rangle$ are identical.

Similarly, for given schemes S and S' we say that

(a) S halts if for every interpretation I of S, $\langle S,I\rangle$ halts;

(b) S diverges if for every interpretation I of S, $\langle S,I\rangle$ diverges;

(c) S and S' are (strongly) equivalent if for every interpretation I of S and S', $\langle S,I\rangle$ and $\langle S',I\rangle$ are strongly equivalent; and

---

† The domains of I and I' are assumed to be equal. Furthermore, S and S' are assumed to have the same set of individual variables. Finally, Value $\langle S,I,\sigma\rangle \cong$ Value $\langle S',I',\sigma\rangle$ iff either
(a) both Value $\langle S,I,\sigma\rangle$ and Value $\langle S',I',\sigma\rangle$ are undefined or
(b) they are both defined and equal.

(d) S and S' are isomorphic if for every interpretation I of S and S', <S,I> and <S',I> are isomorphic.

Definition 2.3: A program scheme S is said to be free if every finite path through its flow diagram is an initial segment of some computation.

Given a scheme S with function variables f_i of arity n_i $i = 1,\ldots,p$, the set of terms of S, denoted Terms(S) is generated by the production

$$\langle term\rangle ::= x_1 | \ldots | x_p$$

$$\langle term\rangle ::= f_i^{n_i}(\langle term\rangle_1,\ldots,\langle term\rangle_{n_i}). \quad i = 1,\ldots,m$$

In the case of Ianov schemes, the set of terms is simply $\{f_1,\ldots,f_m\}^+x$.

A Herbrand interpretation H of scheme S is an interpretation such that

(i) domain of H is Terms(S)

(ii) $(f_i)_I((t_1)_I) := f_i(t_1,\ldots,t_n)$

We also consider containment and weak equivalence which we will not define here.

First, we study the relationship of the complexity of several of the predicates mentioned above to the underlying graph structure of the flow diagrams of the schemes. Our first result, true for all classes of program schemes, has especially strong corollaries when applied to tree schemes.

Lemma 2.4: There is a deterministic polynomial time bounded algorithm M to decide, given a pair (S,Π) where S is a program scheme and Π is a path through S, if Π is executable under some interpretation.

Proof: We need consider only Herbrand interpretations. Π is not executable iff there are two occurrences of some predicate P in such that (i) the values of $x_1,\ldots,x_n$ are the same for both occurrences of P for all Herbrand interpretations of S and (ii) Π takes different branches after the first and second occurrences of P.

For each node n_i in Π record the values of $x_1,\ldots,x_n$ (denoted by $x_1(i),\ldots,x_n(i)$) at that node together with the values of $P(x_1,\ldots,x_n)$ for those predicates P and values of $x_1,\ldots,x_n$ that have been determined by nodes n_1 through n_i of Π. Thus, if the instruction at node n_i is of the form $y \leftarrow f(x_1,\ldots,x_n)$, then for all variables x_j except y, $x_j(i) = x_j(i-1)$. $y(i) = f\ x_1(i-1),\ldots,x_n(i-1)$. (We set $x_j(0) = x_j$ for all individual variables x_j.) Similarly, if the instruction at node n_i is of the form IF $P(x_1,\ldots,x_n)$ THEN GO TO k_1 ELSE GO TO k_2 and node n_{i+1} corresponds to the k_1 [or k_2]$^{\underline{th}}$ instruction of S, then for all individual variables $x_j, x_j(i) = x_j(i-1)$. But, $P(x_1(i-1),\ldots, x_n(i-1))$ must now be true [or false] in order to execute Π up to node n_{i+1}.

For each predicate P_j in Π we need only consider each pair of occurrences of P_j say at nodes n_k and n_ℓ (denoted by $P_{j,k}$ and $P_{j,\ell}$ and verify that the values forced by these occurrences of P_j in Π are consistent, i.e., either $x_1(k) \neq x_1(\ell) \vee \ldots \vee x_n(k) \neq x_n(\ell)$, or $P_{j,k},\ldots,x_n(k)) \neq P_{j,\ell}(x_1(\ell),\ldots,x_n(\ell))$. The number of predicates

$P_j \leq \max(|S|,|\Pi|)$. The number of nodes $n_i \leq |\Pi|$. Thus, the algorithm needs at most time polynomial in $\max(|S|,|\Pi|)$.

The following two results are straightforward corollaries of Lemma 2.4.

Proposition 2.5: There is a deterministic polynomial time bounded algorithm for converting an arbitrary tree scheme into a strongly equivalent free tree scheme.

Proof: For a tree scheme S the number of distinct maximal paths in S is equal to the number of leaves of the underlying flow diagram, which is a tree. Thus, the number of maximal paths $\leq$ number of instructions in S. Using 2.4 prune all unexecutable paths as illustrated below. The remaining tree scheme S' is free and strongly equivalent to S.

Theorem 2.6: For tree schemes there are deterministic polynomial time bounded algorithms for

(i) freedom,

(ii) divergence

(iii) halting,

(iv) isomorphism, and

(v) redundant function, redundant predicate, or redundant loop.

Similarly for free schemes we have

Theorem 2.7: For free program schemes there are deterministic polynomial time bounded algorithms for

(i) divergence,

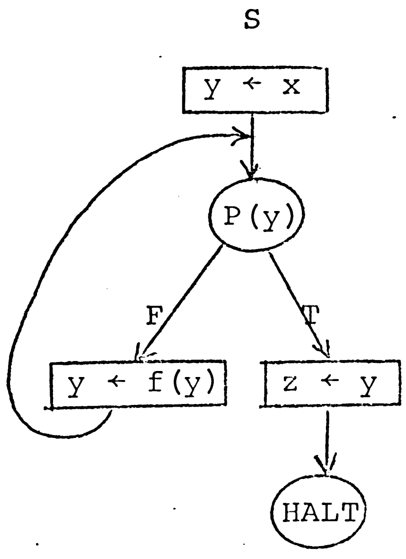
(ii) halting, and

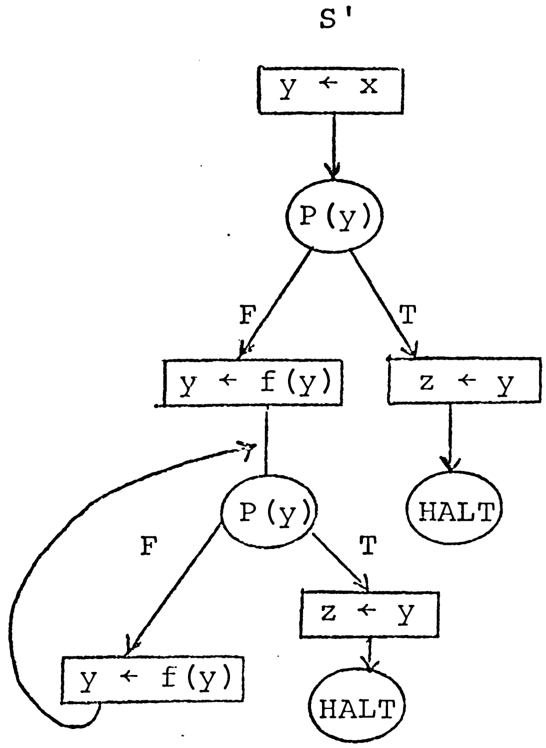
(iii) isomorphism.

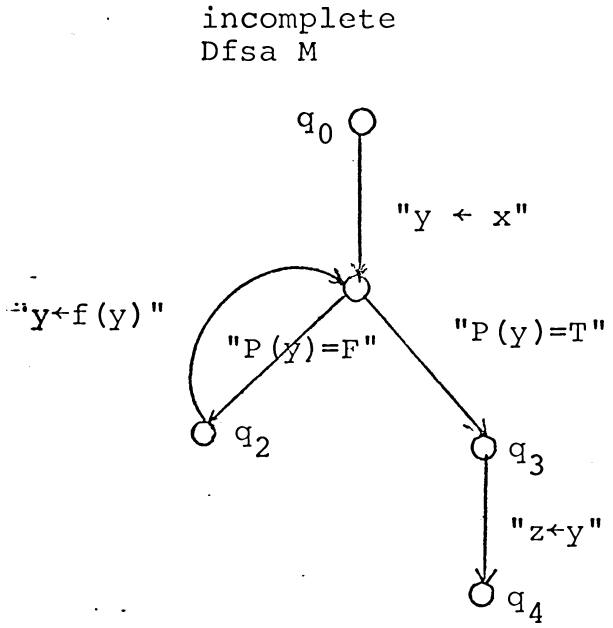
Proof: (i) A free program scheme diverges iff it does not contain a halt instruction.

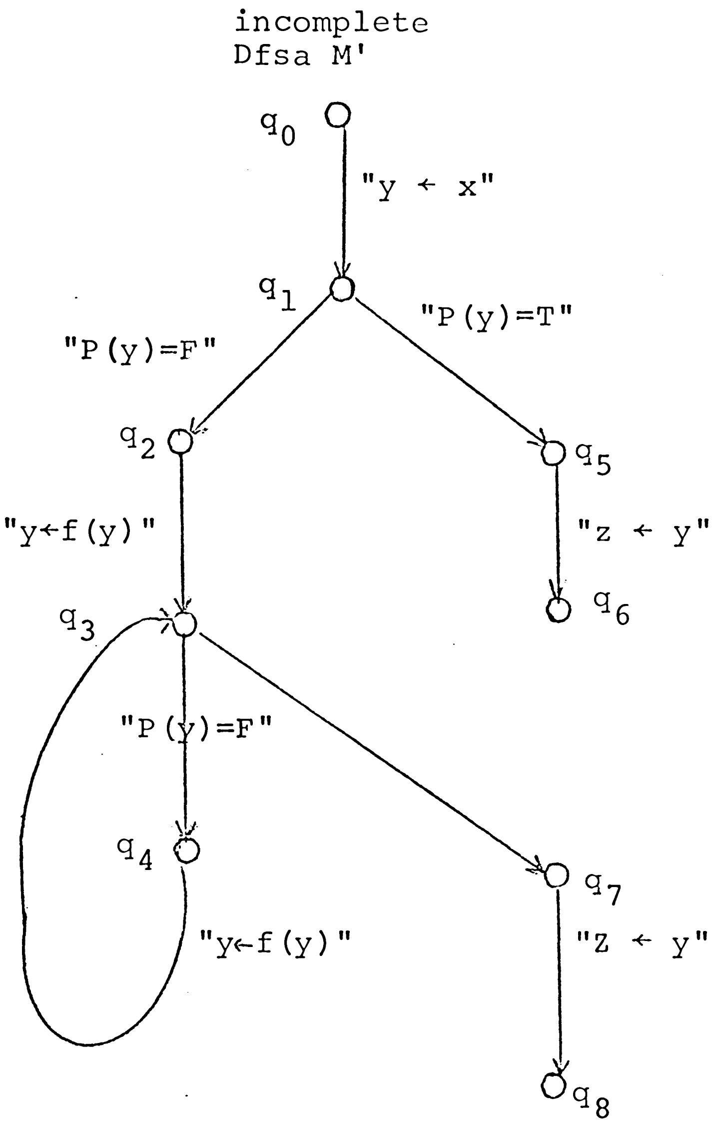
(ii) A free program scheme halts iff it does not contain a loop statement and its flow diagram contains no looping in its graph structure.

(iii) We modify the construction of Kaplan [7].
We illustrate the construction by an example. In §3 we consider the problem in more detail.









But M and M' are incomplete <u>deterministic</u> finite automata. They can be completed without increasing their size by more than a square factor. It is clear that S is isomorphic to S' iff L(M) = L(M').

Thus, given S construct M as follows:

<u>STEP 1</u>: For all nodes n_i in the flow diagram of S there is exactly one state q_i. The start state of M is that state that corresponds to the start statement of S. The accepting states of M are all states q_i that correspond to halt statements. There is one trap state distinct from the

states mentioned above. If the statement at node i of S is $y \leftarrow f(x_1,\ldots,x_n)$ and the next executable node is node j then $\delta(q_i, \text{"}y - f(x_1,\ldots,x_n)\text{"}) = q_j$. If the statement at node i of S is IF $P_k(x_1,\ldots,x_n)$ THEN GO TO k_1 ELSE GO TO k_2 , then

$\delta(q_i, \text{"}P_k(x_1,\ldots,x_n) = T\text{"}) = q_{k_1}$ and

$\delta(q_i, \text{"}P_k(x_1,\ldots,x_n) = F\text{"}) = q_{k_2}$. We leave the remainder of the construction to the reader.

STEP 2: Test the equivalence of M and M' using one of the known deterministic polynomial algorithms, say Hopcroft's $n \log n$ algorithm in [5]. ∎

Unfortunately it is well known that freedom for arbitrary program schemes is undecidable. However, Theorem 2.7 does imply that predicates (i), (ii), and (iii) are deterministic polynomial for all subclasses of schemes which can be shown to be free. In particular,

Lemma 2.8: There is a deterministic polynomial time bounded algorithm to determine if a Ianov scheme is free.

Proof: Immediate from the facts that (i) a monadic single variable scheme is not free iff it has a path in its flow diagram with two identical tests and no assignment statement between them, (ii) only paths of length $\leq$ number of predicates appearing in the scheme need be considered, and (iii) the number of nodes n_j reachable from n_i is $\leq$ the number of nodes of the flow diagram. ∎

Next we present simple sufficient conditions on predicates on monadic single variable schemes which guarantee that any predicate satisfying them is NP-hard. All our results for NP-hard predicates on single variable schemes follow from Proposition 2.11 below. First, we need several definitions:

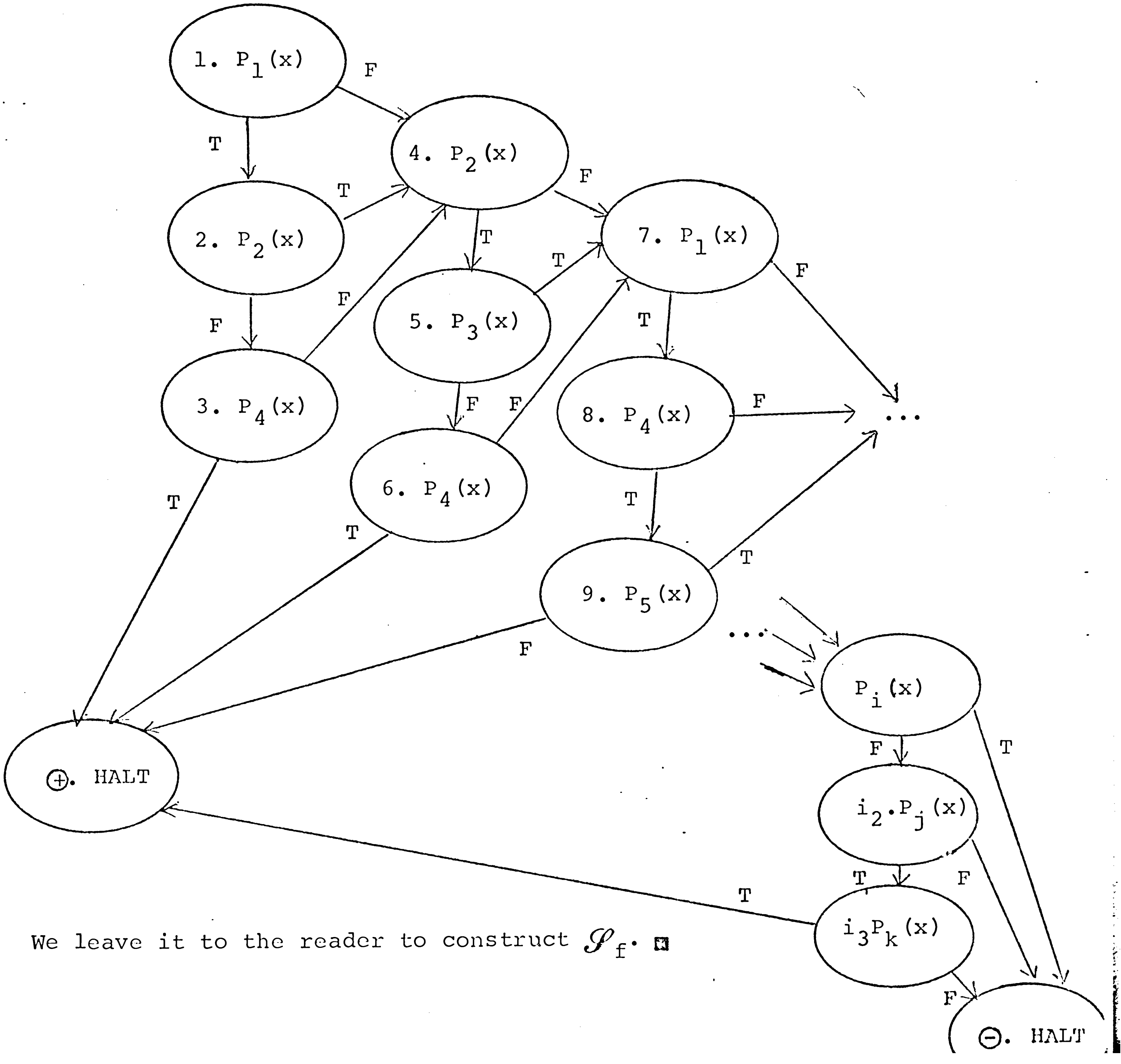
Definition 2.9: A Boolean form is a D_3-Boolean form if f is the disjunction of clauses $c_1,\ldots,c_p$ such that each clause c_i is the conjunction of at most three literals. Similarly, f is a D_2-Boolean form if f is the disjunction of clauses $c_1,\ldots,c_p$ such that each clause c_i is the conjunction of at most two literals. Cook [2] has shown that the set of nontautological D_3-Boolean forms is an NP-complete form. He also has that the set of tautological D_2-Boolean forms $\in$ P.

Definition 2.10: A switching scheme $\mathcal{S}$ is a monadic, single variable, loop-free program scheme such that all its instructions are either conditional or halt instructions. Those statements of that are halt instructions are called terminal statements.

† A scheme whose underlying flow diagram has only nodes of indegree ≤ 1 is a tree scheme provided its flow graph is connected.

Proposition 2.11: There exists a deterministic polynomial time bounded Tm M such that M, when given a D_3-Boolean form f as input, outputs a switching scheme $\mathcal{S}_f$ with exactly two terminal statements labeled $\oplus$ and $\ominus$ such that statement $\ominus$ is reachable by some computation path in $\mathcal{S}_f$ iff f is not a tautology.

Proof: We illustrate the construction of $\mathcal{S}_f$ by an example. Let $f = x_1\bar{x}_2x_4 \vee x_2\bar{x}_3x_4 \vee x_1x_4\bar{x}_5 \vee \ldots$. The flow diagram for $\mathcal{S}_f$ is given below.



We leave it to the reader to construct $\mathcal{S}_f$. ∎

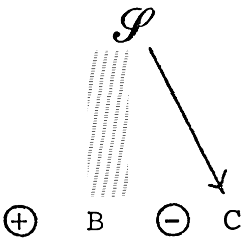
We note that if we allow $\mathcal{S}_f$ to have $O(t)$ terminal statements where t is the number of terms in f, then

(i) the construction of $\mathcal{S}_f$ is still deterministic polynomial time bounded in $|f|$,

(ii) the flow diagram of $\mathcal{S}_f$ is a directed acyclic graph each of whose nodes has in-degree at most 2, and

(iii) $\underset{\sim}{c}$ is reachable by some computation path in $\mathcal{S}_f$ iff f is <u>not</u> a tautology.

<u>Definition 2.12</u>: Let $\mathcal{S}$ be a scheme with exactly two terminal statements labeled ⊕ and ⊖ . Given schemes B and C ,



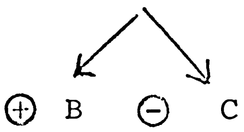
also written $\mathcal{S} \rightarrow B,C$ is the scheme which results by replacing statement ⊕ by B and statement ⊖ by C with some suitable renumbering of the statements in B and C if necessary. For example let $\mathcal{S}$, B and C be as below.

$\mathcal{S}$: 1. IF P_1(X) THEN 2 ELSE 4.
2. X ← X.
3. HALT.
4. X ← X.
5. HALT.

B: 1. LOOP.

C: 1. X ← f(x).
2. HALT.

Then $\mathcal{S}$ is as given below.



1. IF P_1(X) THEN 2 ELSE 4.
2. X ← X.
3. LOOP
4. X ← X.
5. X ← f(x).
6. HALT.

Theorem 2.13: Let Q be any predicate on program schemes such that $\exists$ monadic single variable program schemes B and C such that $\forall$ switching schemes $\mathscr{S}$ with exactly two terminal statements $\oplus$ and $\ominus$, $Q((\mathscr{S} \longrightarrow B,C))$ is true iff $\ominus$ is reachable by some computation. Then $\{\mathscr{S} | \mathscr{S}$ is a monadic single variable flowchart scheme and $Q(\mathscr{S})$ is false$\}$ is NP-hard. Moreover, if B and C are loop-free (have only loop instructions but no looping in their flow diagrams) then $\{\mathscr{S} | \mathscr{S}$ is a loop-free monadic single variable program scheme ($\mathscr{S}$ is a monadic single variable program scheme which has only loop instructions but no looping in its flow diagram) and $Q(\mathscr{S})$ is false$\}$ is NP-hard.

Proof: Let f be an arbitrary D_3-Boolean form. Let $\mathscr{S}_f$ be the switching scheme constructed in 2.11. Let statement $\oplus$ be statement $\oplus$ of $\mathscr{S}_f$. Let statement $\ominus$ be statement $\ominus$ of $\mathscr{S}_f$. Then $\ominus$ is reachable iff f is not a tautology. Since all constructions are deterministic polynomial time bounded the theorem follows. ☐

Corollary 2.14: The following predicates or their negations satisfy the hypotheses of Theorem 2.13:

(i) divergence,

(ii) halting,

(iii) strong equivalence,

(iv) weak equivalence,

(v) containment,

(vi) isomorphism, and

(vii) redundant loop, predicate, or function.

(i), (ii), and redundant loop are NP-complete for multiple variable schemes with loop instructions but no looping in their flow diagram. (iii), (iv), (v), (vi), redundant loop, and redundant function are NP-complete for loop-free multiple variable schemes.

<u>Proof</u>:

1. B is 1. LOOP. C is 1. HALT.

2. B is 1. HALT. C is 1. LOOP.

3. Let B be 1. HALT. Let C be 1. X ← f(x).
   2. HALT.

$$\begin{tikzcd} & \mathscr{P}_f \arrow[dl] \arrow[dr] & \\ +1.\ \text{HALT.} & & -1.\ x \leftarrow f(x). \\ & & -2.\ \text{HALT.} \end{tikzcd}$$

is strongly equivalent to 1. HALT iff −1 is not reachable.

4. For schemes without loop instructions or looping in their computation graphs, strong and weak equivalence are the same.

5.
$$\begin{tikzcd} & \mathscr{P} \arrow[dl] \arrow[dr] & \\ +1.\ \text{HALT} & & -1.\ x \leftarrow f(x). \\ & & -2.\ \text{HALT.} \end{tikzcd}$$

contains 1. HALT iff −1 is not reachable.

6. Let S be
$$\begin{tikzcd} & \mathscr{P}_f \arrow[dl] \arrow[dr] & \\ +1.\ \text{HALT} & & -1.\ \text{HALT} \end{tikzcd}$$

Let S' be $\mathscr{S}_f$

+1'.HALT   −1'. $x \leftarrow f(x)$

−2'. HALT.

Then S is isomorphic to S' iff −1'. is not reachable.

7. <u>Redundant loop</u>: (1) B is 1. HALT and (2) C is 1. LOOP

<u>Redundant function</u>: (1) B is 1. HALT and (2) C is 1. $x \leftarrow f(x)$.
2. HALT.

<u>Redundant predicate</u>: (1) B is 1. HALT and
(2) C is 1. IF $P_0(x)$ THEN
GO TO 2 ELSE
GO TO 2.
2. HALT.

Finally, in each case completeness follows from the absence of looping in the flow diagrams of the schemes and Lemma 2.4. ◻

Theorem 2.13's importance lies in the weakness of the hypotheses needed to guarantee that any predicate satisfying them is NP-hard. Since we used no looping at all except possibly trivial loops and only monadic functions and predicates, these results are strongly upwards reducible, i.e., any class of schemes containing the "complexity core" of 2.12 and 2.13 has at least NP-hard equivalence, halting, divergence, and isomorphism problems (e.g. Paterson's monadic schemes with nonintersecting loops [6]). Also of interest is the relationship between the complexity of these predicates on a class of schemes C and the underlying graph structures of the flow diagrams of the schemes in C. In particular divergence, halting, isomorphism, and freedom for tree schem s are all elements of P. However, they are NP-complete for schem s whose underlying flow diagrams are directed acyclic graphs all of whose nodes have in-degree $\leqq 2$. This is analogous combinatorially to the facts due to Cook mentioned above that the set of nontautological D_3-Boolean forms is NP-complete while the set of nontautological D_2-Boolean forms is an element of P. Moreover, these results are directly embeddable in recursion schemes.

Definition 2.15: A monadic recursion scheme is a finite list of definitional equations

$$F_1X \leftarrow \text{ IF } P_1X \text{ THEN } \alpha_1X \text{ ELSE } \beta_1X$$

$$\cdots$$

$$F_nX \leftarrow \text{ IF } P_nX \text{ THEN } \alpha_nX \text{ ELSE } \beta_nX,$$

where $F_1,\ldots,F_n$ are new defined function symbols; $P_1,\ldots,P_n$ (not necessarily distinct) are predicate symbols; $f_1,\ldots,f_m$ are basis function symbols; and $\alpha_1, \beta_1,\ldots,\alpha_n, \beta_n$ are (possibly empty) strings of defined and basis symbols. A monadic recursion scheme is linear if at most one defined function symbol occurs in each of the strings $\alpha_1, \beta_1,\ldots, \alpha_n, \beta_n$. A monadic recursion scheme is right linear if it is linear and in each string $\alpha_1, \beta_1,\ldots, \alpha_n, \beta_n$ basis function symbols occur only to the right of the defined function symbol (if a defined function symbol is present). For the relationship of the above to context-free grammars see [3]. Again we assume the reader knows the standard results concerning Herbrand interpretations.† □

---

†If not see [1] or [3].

It is well known that monadic single variable program schemes are effectively translatable into strongly equivalent right-linear monadic recursion schemes by assigning to each instruction in the program scheme a defined function F_j and a defining equation for F_j as follows:

(i) if statement j is of the form

$$j. \quad x \leftarrow f(x)$$

then the defining equation for F_j is

$$F_j x \leftarrow \text{IF } P_1 X \text{ THEN } F_{j+1} f\ X \text{ ELSE } F_{j+1}\ f\ X,$$

(ii) if statement j is of the form

$$j. \text{ IF } P_k x \text{ THEN GO TO } k_1 \text{ ELSE GO TO } k_2$$

then the defining equation for F_j is

$$F_j x \leftarrow \text{IF } P_k x \text{ THEN } F_{k_1} X \text{ ELSE } F_{k_2} x, \text{ and}$$

(iii) if statement j is of the form

j. HALT

then the defining equation for F_j is

$$F_j x \leftarrow x \;.$$

We note that this translation can be effected by a deterministic polynomial time bounded Tm. Thus as a corollary to 2.12, 2.13, and 2.14 we have

<u>Theorem 2.16</u>: For right linear monadic recursion schemes and hence for linear monadic recursion schemes, the strong equivalence, weak equivalence, halting, divergence, containment, and redundant defining equation problems or their negations are NP-hard.

Next, we show that several of these predicates are in NP for linear monadic recursion schemes. To do this we need the following result of Garland and Luckham [3]:

<u>Proposition 2.17</u>: Let R and S be two linear monadic recursion schemes such that

(i) R and S have at most e recursion equations,

(ii) each α_i, β_i in a defining equation of R or of S contains at most ℓ function symbols, and

(iii) in any defining equation $F_j x \leftarrow$ IF Px THEN $\alpha_j x$ ELSE $\beta_j x$, $|\alpha_j|, |\beta_j| \leq 1$.

Then $\exists$ a Herbrand interpretation I under which R and S differ such that n, the minimum of the lengths of $\text{Value}_I(R)$ and $\text{Value}_I(S)$ (at least one of which is defined), is $< 3e^3\ell$.

For a proof see [3].

<u>Theorem 2.18</u>: The negations of the strong equivalence and divergence problems for linear monadic recursion schemes are NP-complete.

<u>Proof</u>: They are both NP-hard by Theorem 2.66. We show that $\{(R,S) \mid$ R and S are linear monadic recursion schemes and R is not strongly equivalent to S$\} \in$ NP. We observe that, without loss of generality, we can assume that each time an input is changed, its length is increased by exactly one basis function symbol. For example, an equation $Fx \leftarrow \alpha\, F'\, f_1 \ldots f_i x$ can be replaced by equations $F \leftarrow F^i f_i$, $F^i \leftarrow F^{i-1} f_{i-1}, \ldots, F^2 \leftarrow \alpha\, F'\, f_1$, where $F^i, \ldots, F^2$ are newly defined function symbols. All such replacements are deterministic polynomial time bounded.

Without loss of generality assume $\mathrm{Value}_I(R)$ is defined. The nondeterministic polynomial time bounded algorithm proceeds as follows:

<u>STEP1</u>: Guess $\mathrm{Value}_I(R) = f_{i_n} \ldots f_{i_1} x$, where $f_{i_1}, \ldots, f_{i_n}$ are basis function symbols and $n < 3e^3 \ell$. Also for $\beta_0 = x$, $\beta_1 = f_{i_1} x$, $\ldots, \beta_n = f_{i_n} \ldots f_{i_1} x$ guess the values of $P_1(\beta_1), \ldots, P_m(\beta_1)$, $\ldots$, $P_1(\beta_n), \ldots, P_m(\beta_n)$, for all predicates $P_1, \ldots, P_m$ in R or S. This completely determines all information of interpretation I used to find $\mathrm{Value}_I(R)$.

<u>STEP2</u>. Verify that under I determined in STEP 1
$\mathrm{Value}_I(R) = f_{i_n} \ldots f_{i_1} x$.

<u>STEP3</u>. Verify that under I determined in STEP1
$\mathrm{Value}_I(S) \not\stackrel{\sim}{=} f_{i_n} \ldots f_{i_1} x$.

At most ℓ equations can be executed without changing the length of an intermediate string of the form $\alpha F \beta$, where α and β are strings of basis function symbols and F is a defined function symbol, without getting into an infinite loop. Similarly, the length of intermediary strings $\alpha F \beta$ can be changed at most n times. Thus, since each application of a defining equation either leaves α and β unchanged and replaces F by F', or changes $\alpha F \beta$ to $\alpha' F' \beta'$, where $|\alpha| + |\beta| \geq |\alpha| + |\beta| + 1$, the total number of function evaluations in STEPS2 and $3 \leq 2 \cdot n(\ell + 1)$.

As an immediate corollary we have

<u>Corollary 2.19</u>: For monadic single variable flowchart schemes (i.e. Ianov schemes), the negations of the strong equivalence and divergence problems are NP-complete.□

Analogous to Theorem 2.13 for multiple variable schemes the following holds.

Theorem 2.20: If $\exists$ two multiple variable schemes A and B such that for all multiple variable flowchart schemes $\mathscr{S}$ with exactly two terminal statements labeled $\oplus$ and $\ominus$, $P(\mathscr{S} \longrightarrow A,B))$ is true iff statement $\oplus$ is never executed and is false otherwise, then $\{ \mathscr{S} \mid P(\mathscr{S}) \text{ is true}\}$ is not r.e.

Proof: From [9] we know that for all Turing machines M and for all initial configurations X of M , we can effectively find a multiple variable scheme $\mathscr{S}_{M,X}$ with exactly two halt instructions labeled $\oplus$ and $\ominus$ such that $\oplus$ is reachable iff M halts on X. Thus $P(\mathscr{S}_{M,X})$ is true iff M does not halt on X . Hence, $\{ \mathscr{S} \mid P(\mathscr{S}) \text{ is true}\}$ is not r.e.☒

We leave to the reader the verification that strong and weak equivalence, divergence, and containment satisfy 2.20. Moreover, any form of Paterson's "reasonable equivalence", see [9], also satisfies 2.20. Theorem 2.20 shows that divergence, containment, and any form of "reasonable equivalence" are undecidable for the same reason. However, neither halting nor its negation satisfy the conditions of 2.20; but halting does satisfy 2.13. This is because the switching schemes of 2.13 are total, while the schemes of 2.20 are generally not total. In any case, informally, 2.13 and 2.20 show that any predicate on flowchart schemes whose truth or falsity depends upon

(i) whether a particular instruction is ever executed or

(ii) whether every terminating computation ultimately executes a particular instruction

is NP-hard for single variable schemes and undecidable for multiple variable schemes.

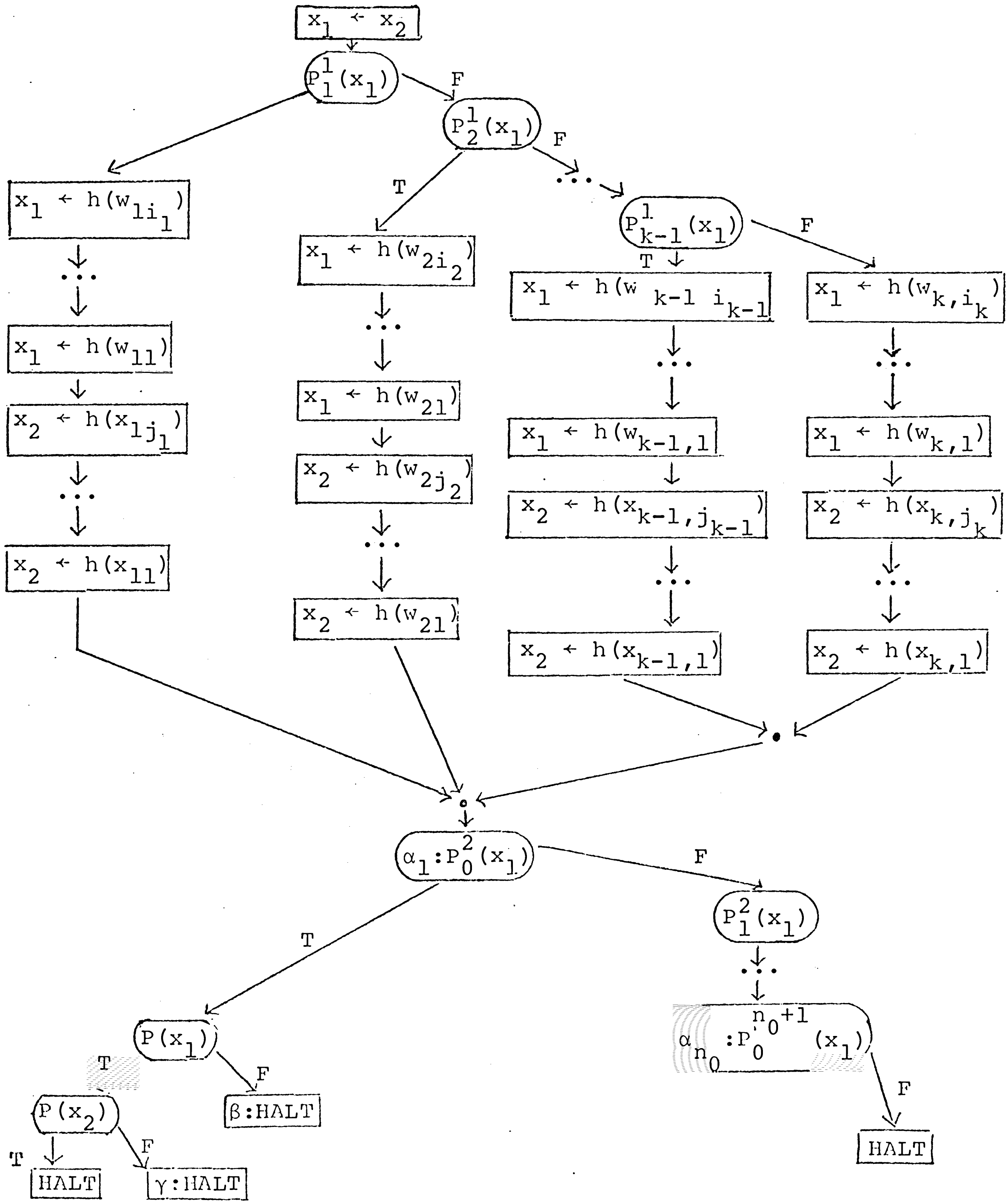
The reader should also note that freedom does not satisfy either (i) or (ii) . As shown by Paterson [9] or Manna [5], freedom is undecidable because Post's Correspondence Problem is embeddable in it. Analogously for 2 variable loop-free monadic schemes we have

Proposition 2.21: The negation of freedom for 2 variable loop-free schemes is NP-complete.

Proof: We leave to the reader to verify (see Hopcroft and Ullman [4], pp. 212-218) that for sufficiently large finite alphabets and for sufficiently large polynomials $p(x)$ the following variant of Post's Correspondence Problem called the polynomial PCP is NP-complete.

Let A and B be two lists of strings in Σ^+, with the same number of strings in each list, say $A = w_1,\ldots,w_k$ and $B = x_1,\ldots,x_k$ where $w_1 = w_{11}\ldots w_{1i_1},\ldots,w_k = w_{k1},\ldots,w_{ki_k}$, $x_1 = x_{11},\ldots,x_{1j_1},\ldots,$ and $x_k = x_{k1},\ldots,x_{kj_k}$. Then the polynomial PCP has a solution iff there is a sequence of integers $i_1,\ldots,i_m$ with $m \leq p(\max(|w_1|+\ldots+|w_k|,|x_1|+\ldots+|x_k|)$ such that $w_{i_1}\ldots w_{i_n} = x_{i_1}\ldots x_{i_m}$.

We construct the flow diagram F for a 2 variable loop-free monadic program schem $\mathscr{S}$ such that all paths in F are executable iff the polynomial PCP above has no solution. Let $\Sigma = \{\sigma_1,\ldots,\sigma_\ell\}$. To each element of Σ, σ_i, we associate a distinct function symbol denoted by $h(\sigma_i)$. Let $n_0 = p(\max(|w_1|+\ldots+|w_k|,|x_1|+\ldots+|x_k|)$. Then F is as constructed below.

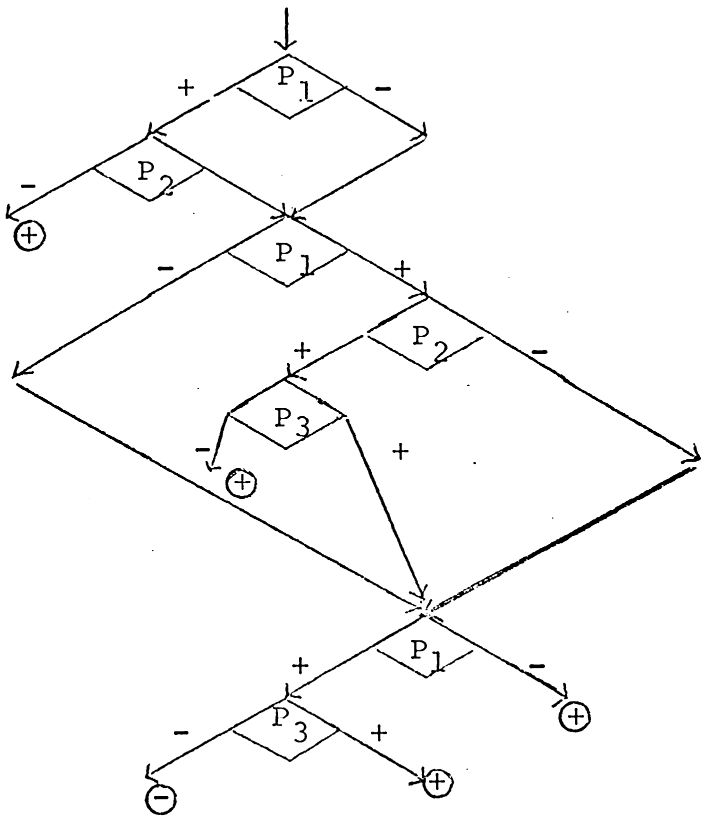

$x_1 \leftarrow x_2$
$P_1^1(x_1)$
F
$P_2^1(x_1)$
F
T
$P_{k-1}^1(x_1)$
F
T
$x_1 \leftarrow h(w_{1i_1})$
$x_1 \leftarrow h(w_{11})$
$x_2 \leftarrow h(x_{1j_1})$
$x_2 \leftarrow h(x_{11})$
$x_1 \leftarrow h(w_{2i_2})$
$x_1 \leftarrow h(w_{21})$
$x_2 \leftarrow h(w_{2j_2})$
$x_2 \leftarrow h(w_{21})$
$x_1 \leftarrow h(w_{k-1\ i_{k-1}})$
$x_1 \leftarrow h(w_{k-1,1})$
$x_2 \leftarrow h(x_{k-1,j_{k-1}})$
$x_2 \leftarrow h(x_{k-1,1})$
$x_1 \leftarrow h(w_{k,i_k})$
$x_1 \leftarrow h(w_{k,1})$
$x_2 \leftarrow h(x_{k,j_k})$
$x_2 \leftarrow h(x_{k,1})$
$\alpha_1 : P_0^2(x_1)$
F
$P_1^2(x_1)$
$\alpha_{n_0} : P_0^{n_0+1}(x_1)$
F
HALT
T
$P(x_1)$
T
F
β:HALT
$P(x_2)$
T
F
HALT
γ:HALT

If there exists some solution of the PCP of length $i \leq i_0$, then the corresponding path in F ending in $\alpha_i: P_0^{i+1}(x_1), P(x_1), P(x_2)$, γ:HALT is not executable. ☒

Finally we note that the polynomial PCP can be used to show that a variety of decidable problems dealing with context-free grammars are intractable, see [7].

§3 Equivalence algorithm for free Ianov schemes

Now that we know that the equivalence problem for Ianov schemes is NP-hard, we inquire whether any interesting subclass of Ianov schemes has a polynomial time bounded equivalence problem. We notice that the proof of NP-hardness involves sieves of predicates of the type



That is, (a) some predicate, such as P_1, P_2 or P_3 above, tests the same value twice so that not all paths are executable; and (b) the sieve is not a tree but a directed acyclic graph (dag).

The first feature of predicate test is unlikely to occur in a well-written program since the programmer would know the outcome of certain tests (e.g. second occurrences of P_1) and would consider those tests unnecessary. Thus we are led to consider program schemes which avoid such predicates. Namely we are interested in Ianov schemes in which no predicate tests the same value more than once. These are the free Ianov schemes. Our question is, "Do free Ianov schemes have a decidable equivalence problem?"

In this paper we only answer part of the question. We show that strongly free Ianov schemes have a polynomial time decidable equivalence problem and that strongly free Ianov schemes in which predicates are replaced by tree-like predicate clusters also have a polynomial time decidable equivalence problem.

In a strongly free scheme there is a function application between any two predicates. These are interesting because they behave like finite automata; in fact, the

technique for showing that equivalence of strongly free Ianov schemes can be decided in polynomial time is to consider such schemes as (deterministic) finite automata. However, there is one serious difficulty. A scheme S may have redundant predicates, i.e. predicates such that the left branch is equivalent to the right branch. (see Examples 3.4, 3.5). This redundancy appears at first sight difficult to detect (see Example 3.4(2)) because equivalence of the branches may involve strings in which the predicate p itself appears. But there is a simple exponential time algorithm to test for redundancy.

To obtain a polynomial time equivalence algorithm we must find a polynomial time redundancy test. This is done by modifying the usual state minimization algorithm for finite automata (Theorem 3.2).

To begin the technical account we introduce the tool of value languages, $L(S)$ and $L^{\#}(S)$. These are languages which describe the possible outputs of a scheme ($L(S)$) and the possible computation sequences ($L^{\#}(S)$). They were used extensively in Garland & Luckham [3].

We then prove theorems relating value languages and equivalence. We are not able to use exactly the methods of Garland & Luckham [3] because of computational complexity considerations.

Recall that $val(S,H,L_0)$ is the value of scheme S under interpretation H starting with statement L_0 , and $val(S,H)$ is the value under H starting at the start statement.

Definition 3.1: The value language of scheme S, denoted L(S) is $\{val(S,H) \mid H \text{ is a Herbrand interpretation}\}$. Thus L(S) is the set of all possible output terms. For Ianov schemes we usually leave off the input variables, thus $L(S) := \{a \mid ax = val(S,H)$ for S Ianov and H Herbrand$\}$

Definition 3.2: A computation of a scheme is a sequence of function applications and predicate evaluations listed in the order performed. A precise definition of a computation for general program schemes is cumbersome, and since we need the concept only for Ianov schemes, we define it only for them as follows:

Comp(S,H) is the unique string $\ldots\alpha_{n+1}P_{i_m}^{\pm}\ldots P_{i_2}^{\pm}\alpha_2 P_{i_1}^{\pm}\alpha_{i_1}$ such that $\alpha_i \in \mathrm{Term}(S)^{\dagger}$ and $P_{i_j}^{\pm}$ are signed predicates of S (i.e. the predicate plus its sign, either P_i^+ or P_i^-) and $val(S,H) = \ldots\alpha_n\ldots\alpha_2\alpha_1$ and $(P_{i_j})_I((\alpha_j\ldots\alpha_1)_I)$ is true iff $P_{i_j}^{\pm}$ is $P_{i_j}^{+}$.

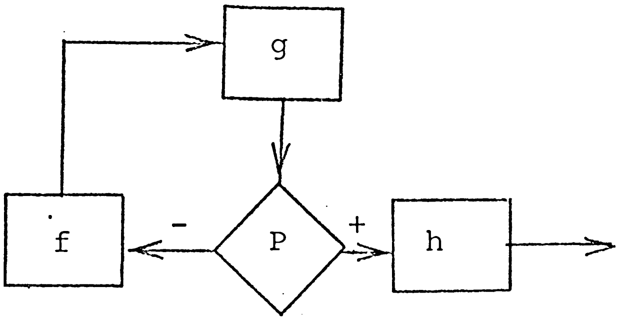
---

†Note, the empty string is a term of S.

$L^C(S) := \{Comp(S,H) | H \text{ is Herbrand}\}$

$L^{C\#}(S) := \{y | y \text{ is finite, } y = Comp(S,H), H \text{ Herbrand}\}.$

Example 3.1: Using regular expressions we describe L(S) and $L^C(S)$ for the scheme S below.



S

$L(S) := \{h(gf)^*g\}$

$L^{C\#}(S) := \{hp^+(gfp^-)^*g\}$

$L^C(S) := \{L^{C\#}(S) \cup \{^{\omega}(gfp-)g\}$

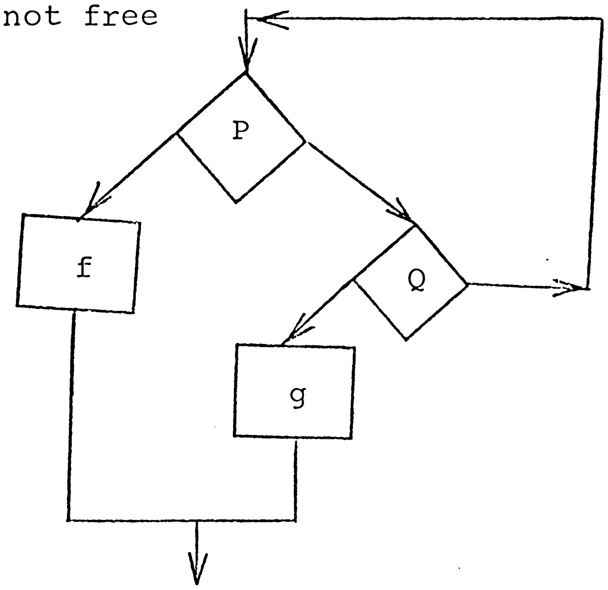
where $^{\omega}(gfp^-)$ is $\ldots gfp^-gfp^- \ldots gfp^-g$.

Recall in Definition 2.3, that a scheme S is free iff no predicate is tested twice with the same value under any Herbrand interpretation. This means there must be a function application between separate occurrences of the same predicate. We now define a similar but stronger notion.

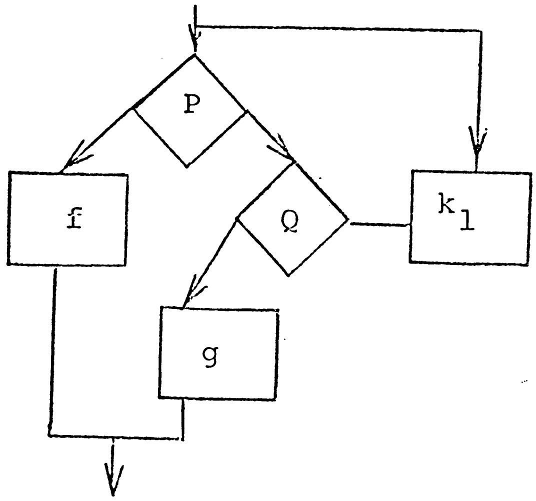
Definition 3.3: A scheme S is strongly free iff no two predicates test the same value in a Herbrand interpretation. Thus there must be a function application between any two conditionals.

Examples 3.2:

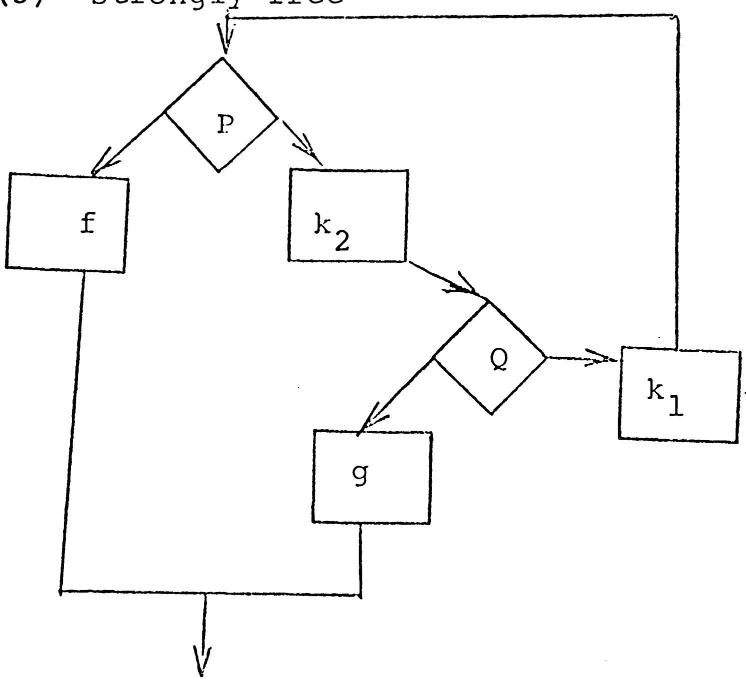
(1) not free



(2) free but not strongly free



(3) strongly free



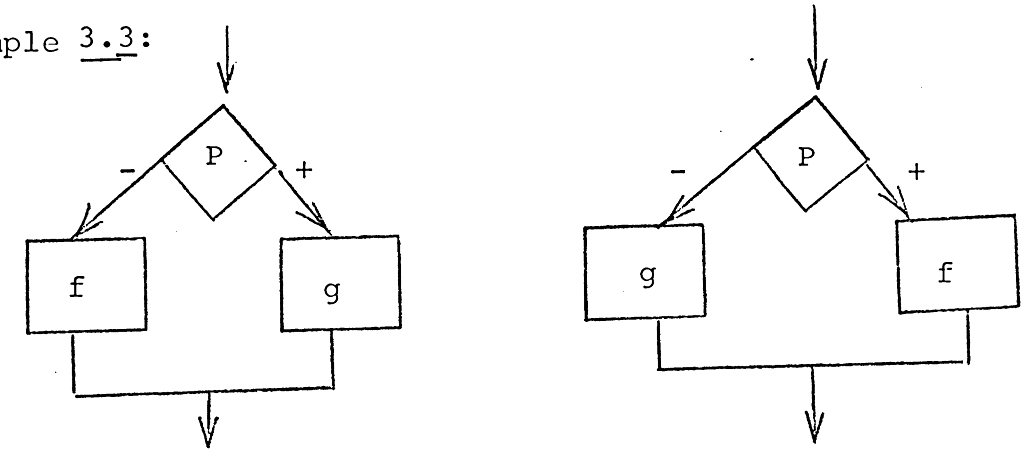
For any class of schemes: Ianov, program, monadic recursion, etc., it is a fact that of two schemes S_1 and S_2 are strongly equivalent, $S_1 \equiv S_2$, then their value languages are the same, $L(S_1) = L(S_2)$. For future reference we cite this as a proposition.

Proposition 3.1: For schemes S_1, S_2

$$S_1 \equiv S_2 \Rightarrow L(S_1) = L(S_2).$$

The converse of this proposition does not hold even for Ianov schemes as the following example shows.

Example 3.3:



This example suggests that some form of converse to the proposition might hold if the value language included information about the order of predicates evaluation. Such a language is $L^c(S)$. For general schemes , say even 2 variable program schemes $S_1 \equiv S_2$ does not imply $L^c(S_1) = L^c(S_2)$ because the schemes may compute the same terms in very different ways (one of them working on one variable first the other on the other variable first). However for any class of schemes we have the converse.

Proposition 3.2: For any schemes S_1, S_2

$$L^c(S_1) = L^c(S_2) \Rightarrow S_1 \equiv S_2 .$$

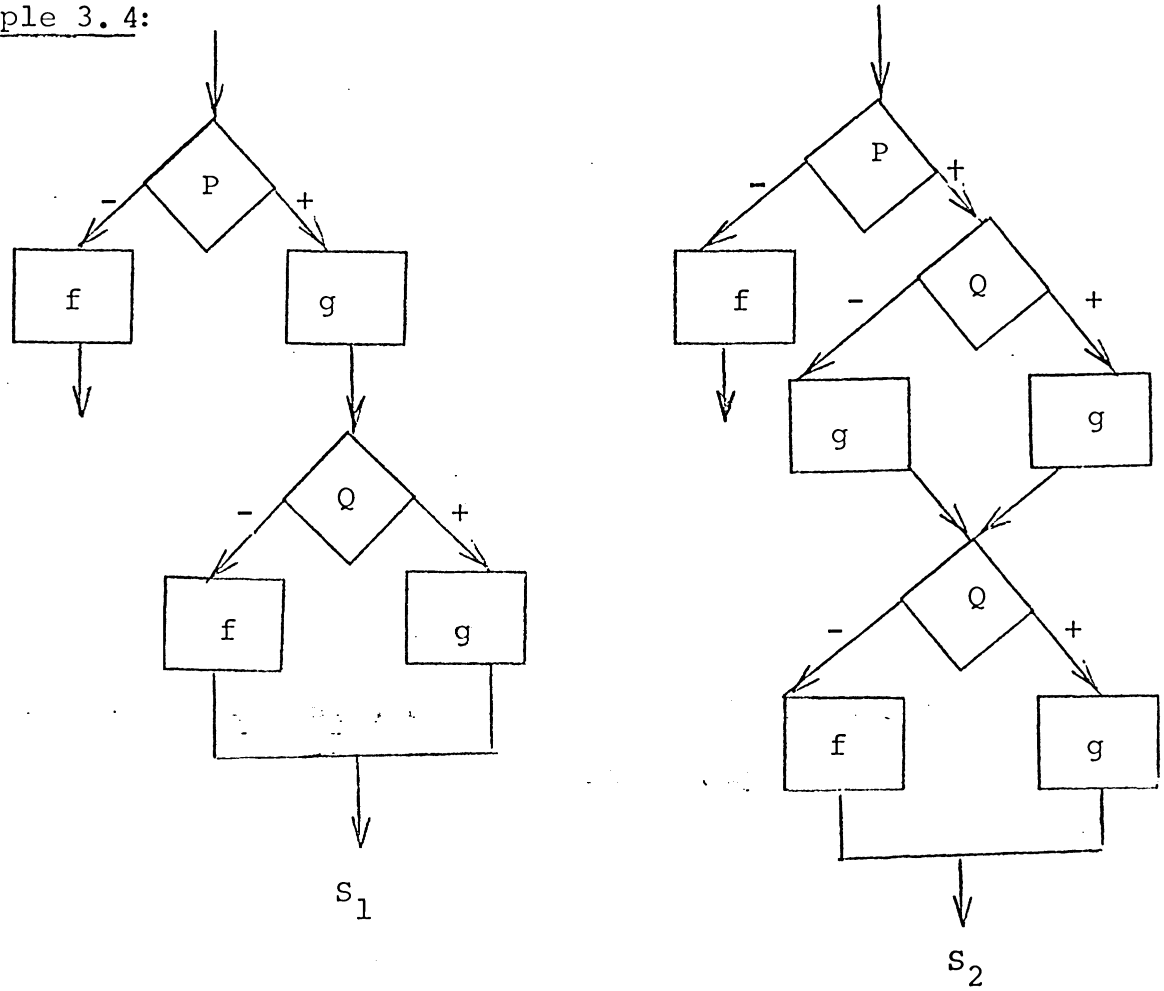
From the discussion we are led to suspect that for one variable schemes we have both propositions, that is

$$* \quad S_1 \equiv S_2 \iff L^c(S_1) = L^c(S_2) .$$

This is almost true, but consider the following examples.

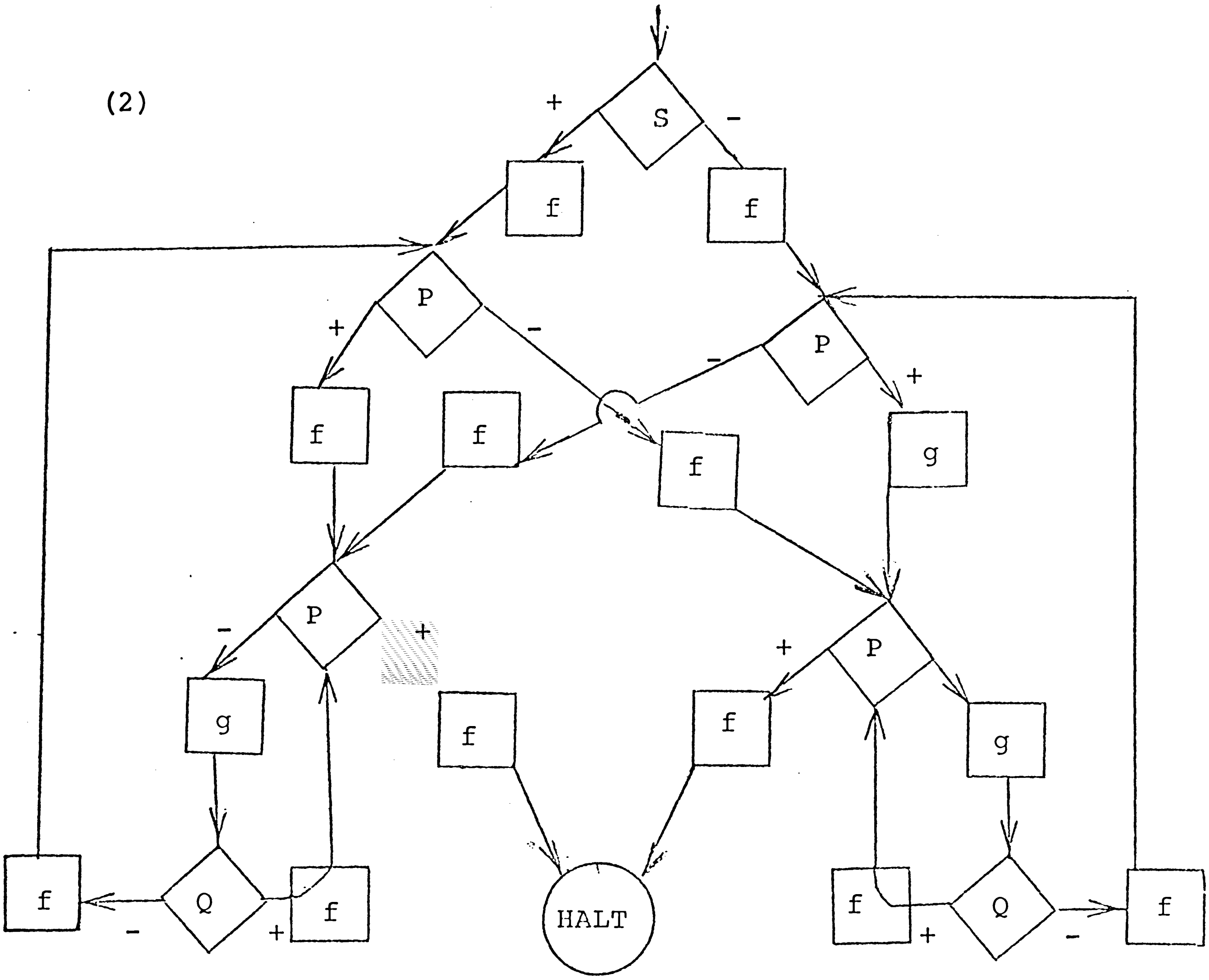
Example 3.4:

(1)

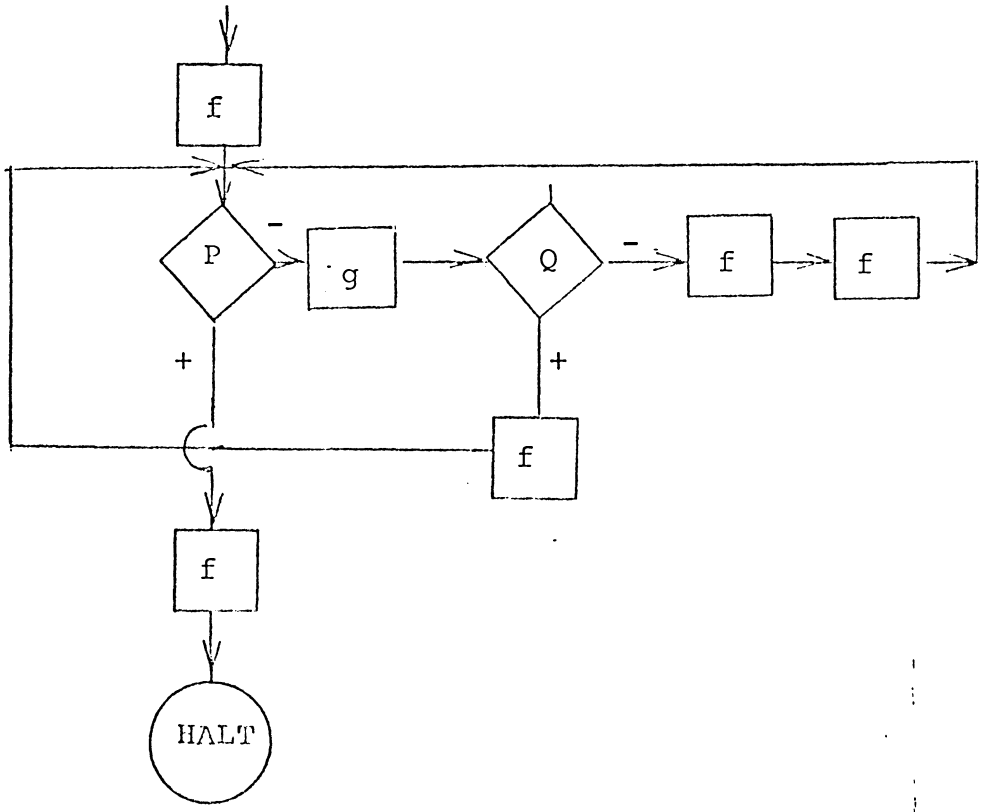


Clearly $S_1 \equiv S_2$, but $L^c(S_1) \neq L^c(S_2)$ because the predicate Q appears in a superfluous way. These superfluous predicates can be more disguised as (2) shows.

(2)



S_1



S

A relationship such as $S_1 \equiv S_2 \iff L^c(S_1) = L^c(S_2)$ is important in designing an equivalence test for Ianov schemes. So we pursue conditions under which the relationship holds. First we find a set of sufficient conditions for a polynomial time equivalence algorithm (Theorem 3.1), namely that there are no superfluous predicates. We then strengthen the result by testing efficiently for superfluous predicates. Finally we generalize to strongly free Ianov schemes.

Example 3.4 shows that $S_1 \equiv S_2 \iff L^c(S_1) = L^c(S_2)$ fails because of the presence of superfluous predicates; they are like noise in the system. So we consider schemes without all the noise.

Definition 3.4: An occurrence of a predicate P (at L1) in scheme S, say

L1: IF P(X) THEN L_ℓ ELSE L_R

is called superfluous iff $val(S,H,L_\ell) = val(S,H,L_R)$ for all Herbrand interpretations H. More generally, two statements L_i, L_j are called equivalent in S iff $val(S,H,L_i) = val(S,H,L_j)$ for all Herbrand H. Thus a predicate occurrence is superfluous iff the two exists are equivalent. A scheme S is called reduced iff there are no equivalent statments.

We can easily show that reduced strongly free Ianov schemes behave like finite automata and that they are characterized by their computation language.

Theorem 3.1: If S_1 and S_2 are reduced strongly free Ianov schemes, then $S_1 \equiv S_2 \iff L^c(S_1) = L^c(S_2)$.

Proof:

(1) (=>) Suppose $S_1 \equiv S_2$, then we show $L^c(S_1) = L^c(S_2)$. For a proof by contradiction, assume $L^c(S_1) \neq L^c(S_2)$. Then let $x = \ldots x_n x_{n=1} \ldots x_2 x_1$ be a string in one language but not in the other, say $x \in L^c(S_1) - L^c(S_2)$. Let x' be the subsequence of x obtained by deleting all predicate tests. Find a sequence y in $L^c(S_2)$ such that (a) deleting all predicate tests (giving y') results in x' , i.e. y' = x'. (There must be such a sequence because $S_1 \equiv S_2 \Rightarrow L(S_1) = L(S_2)$ by Proposition 3.1 and (b) no other sequence in $L(S_2)$ satisfies (a) and agrees with x on a longer initial segment as (from right to left).

Let y_k be the first place where $x \neq y$. This must be a predicate test since x' = y'. Suppose the test is P_i in x and P_j in y appearing as

L_i: IF P_i THEN L_1 ELSE L_2
L_j: IF P_j THEN $\bar{L}_2$ ELSE $\bar{L}_2$

Then L_1 can not be equivalent to both $\bar{L}_1$ and $\bar{L}_2$ otherwise P_j would be superfluous. So suppose L_1 and $\bar{L}_1$ are not equivalent. Then for any w we can choose a Herbrand interpretation H( )† such that

$$\mathrm{val}(S_1, H(w), L_1) \neq \mathrm{val}(S_2, H(w), \bar{L}_1).$$

We can also choose a finite interpretation H_0 which leads to x_k in S_1 and y_k in S_2. If we let δ_{S_1} , δ_{S_2} denote the state

†By H(w) we mean the interpretation in which the input variable has value w.

transition functions and L_0 , $\bar{L}_0$ denote the start states, then we can symbolize this as

$$\delta_{S_1}(H_0,L_0) = x_k$$

$$\delta_{S_2}(H_0,L_1) = y_k$$

We can now extend the inteppretation of H_0 by taking w as $val(S_1,H_0,L_0)$ (which is equal to $val(S_2,H_0,\bar{L}_0)$. Then

$$val(S_1,H(val(S_1,H_0,L_0)),L_0) \neq val(S_2,H(val(S_2,H_0,\bar{L}_0),\bar{L}_0).$$

Hence $S_1 \not\equiv S_2$. Therefore since we assumed $S_1 \equiv S_2$, it must be that $L^c(S_1) = L^c(S_2)$.

(2) ($\Leftarrow$) Now assume $L^c(S_1) = L^c(S_2)$. We then show that $S_1 \equiv S_2$. Suppose that $S_1 \not\equiv S_2$. Then there is some Herbrand interpretation, H_0 (see [3] for a proof that Herbrand interpretations suffice). such that $val(S,H_0) \neq val(S_2,H_0)$. Let the computation of S_1 on H_0 be x and on S_2 be y. Then $x \neq y$.

Since $L^c(S_1) = L^c(S_2)$, the computation x must occur in $L^c(S_2)$. Suppose it occurs under interpretation H_1. We claim that it must also occur for interpretation H_0 because in fact H_0 must be the same as H_1. This is because H_1 is determined by the computation x since the scheme S_1 is deterministic. Thus it must be that $x = y$ so $val(S_1,H) = val(S_2,H)$ for all H.□

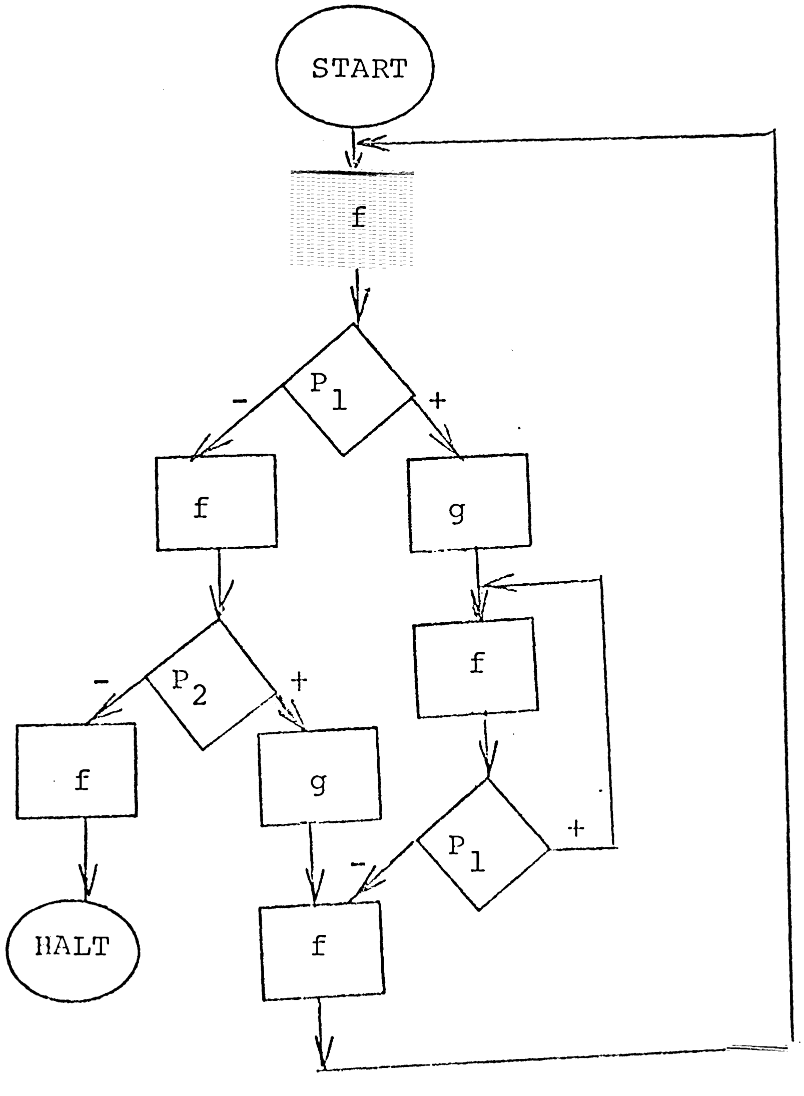
<u>Proposition 3.3</u>: For reduced strongly free Ianov schemes S_1, S_2, $S_1 \equiv S_2 \iff L^{c\#}(S_1) = L^{c\#}(S_2)$.

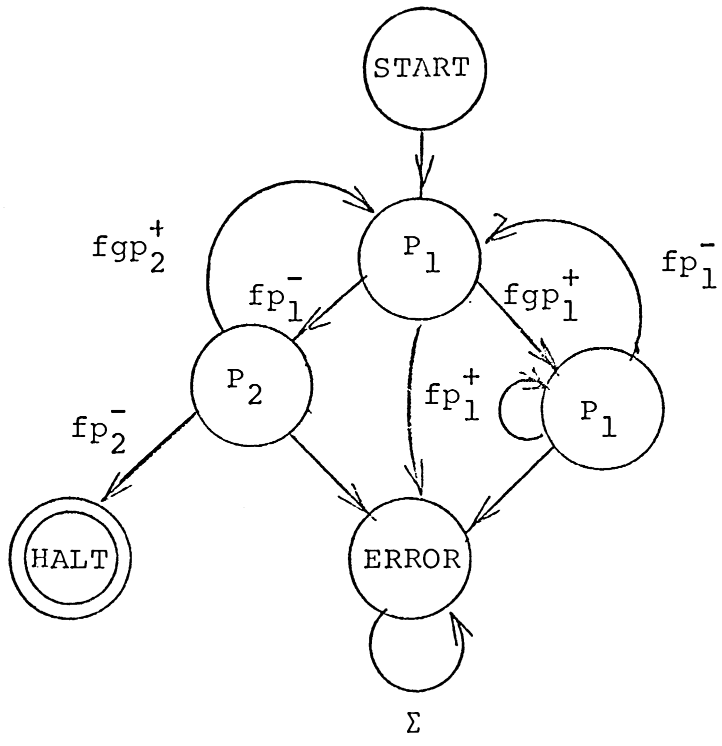
Proof: Clearly we need only show <= .

We need only consider the case where the computation of one S_i is infinite. In that case the other scheme on the same input must also be infinite because if it were finite then by the reasoning in the previous theorem and the assumption $L^{c\#}(S_1) = L^{c\#}(S_2)$ we have that the first computation must be finite. □

Remark: We can view a strongly free scheme S (reduced or not) as a finite automaton accepting $L^c(S)$ or as a finite state transducer generating the set $L^c(S)$. To see how this works, consider the following simple example of a scheme S and its associated finite automaton, A(S).

Example 3.5:





A(S)

The alphabet of the automaton is

$$\sum := \{f,\ fp_1^-,\ fgp_1^+,\ fp_1^+,\ fp_2^-,\ fgp_2^+\}$$

The state set is

$$K := \{\text{start},\ P_1^1,\ P_2^1,\ P_1^2,\ \text{halt},\ \text{error}\}$$

where P_i^j is the j-th occurrence of predicate P_i (in some arbitrary method of ordering occurrences).

In this diagram we used an error state and intended that the unlabeled edges be implicitly labeled by those elements from $\sum$ not occurring as labels on outgoing edges.

Another way to handle the fact that not every state has a meaningful transition under each element of $\sum$ is to leave the diagram incomplete, that is, allow the transition function to be undefined at certain inputs $\langle k,a \rangle$ $k \in K$, $a \in \sum$.

Notice that automaton $A(S)$ accepts precisely the set $L^c(S)$. Thus using the standard notation between automata and their acceptors (see Hopcroft & Ullman [6]) we have

$$L^c(S) = T(A(S)) := \{\text{tapes accepted by } A(S)\}.$$

Thus

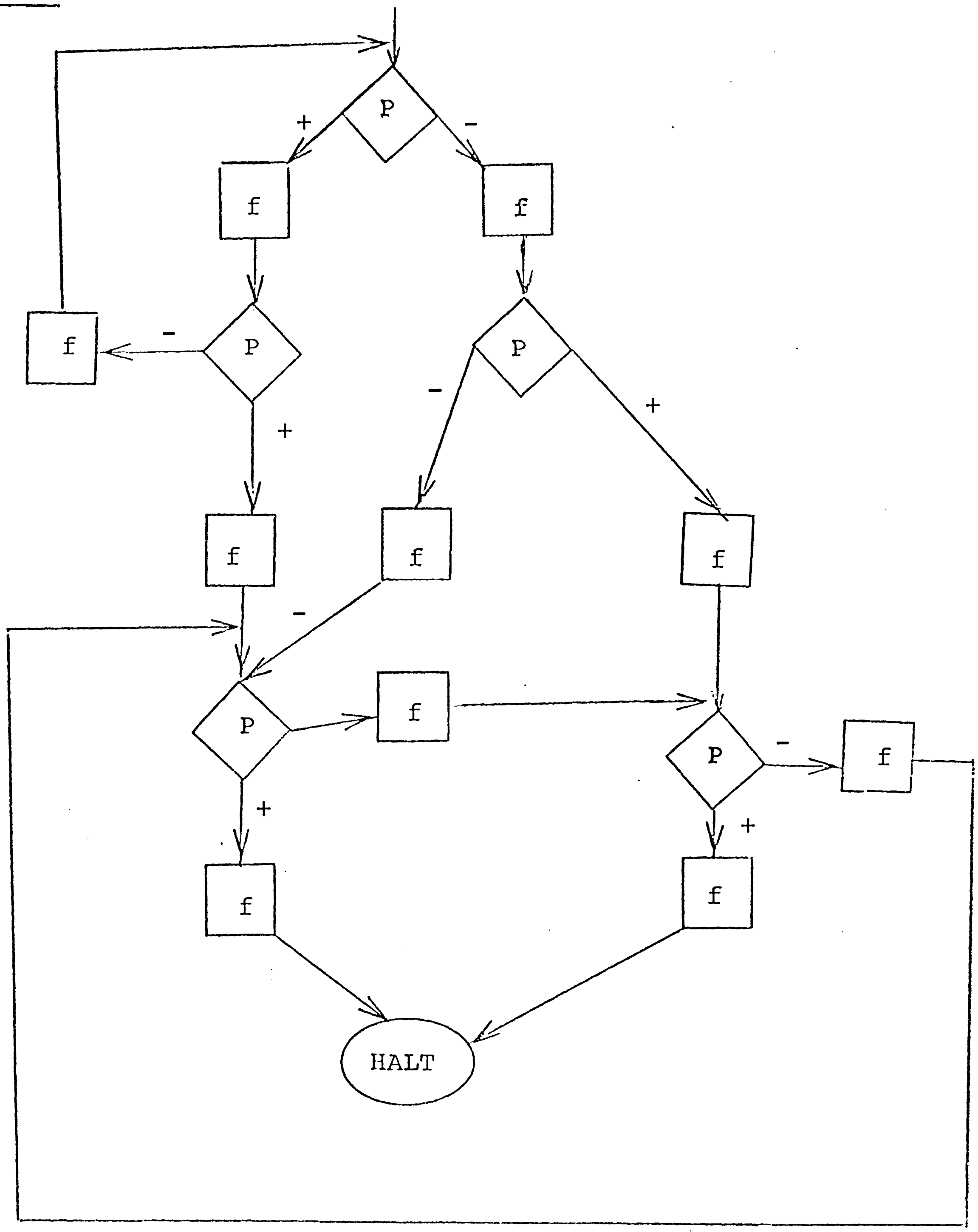
$$S_1 \equiv S_2 \quad \text{iff} \quad A(S_1) \equiv A(S_2).$$

That is, two schemes are strongly equivalent iff the two associated automata accept the same sets.

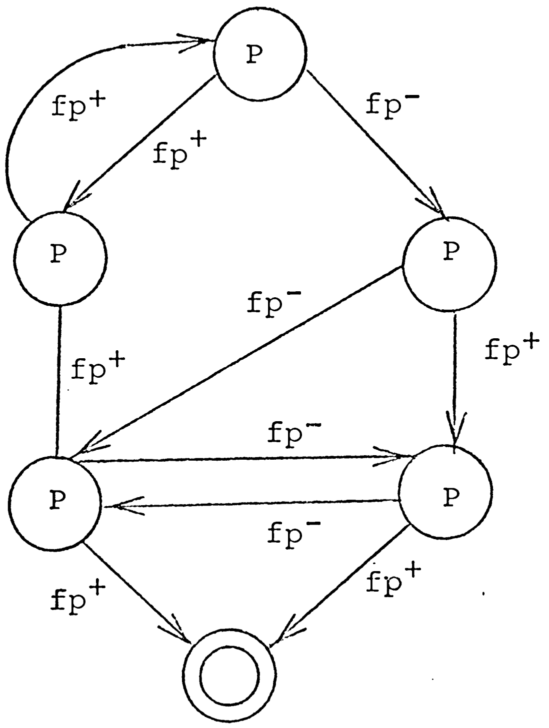
Remark: Notice that this relationship between automata and schemes would fail if we did not use the signed predicates such as P_1^+, P_1^-, etc. as labels. Also notice that for relating the usual concept of finite automaton acceptance to scheme equivalence, it is important to have $L^c(S)$ consist only of the terminating computations. However from the scheme viewpoint it is more convenient to have $L^c(S)$ include the infinite computation as well. Our trivial proposition 3.3 shows that it does not matter which definition we use.

In order to extend the equivalence algorithm to arbitrary strongly free Ianov schemes, we give a method of reducing such schemes. We can not simply regard these schemes S as finite automata $A(S)$ and then reduce $A(S)$. The difficulty is illustrated by this simple example.

Example 3.6:

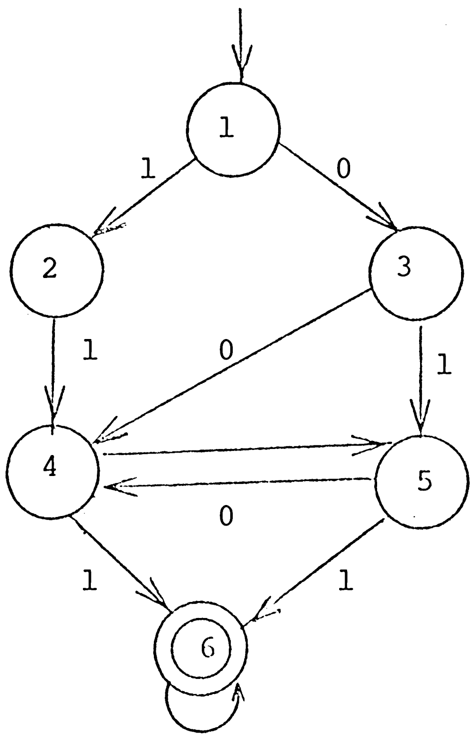


scheme S



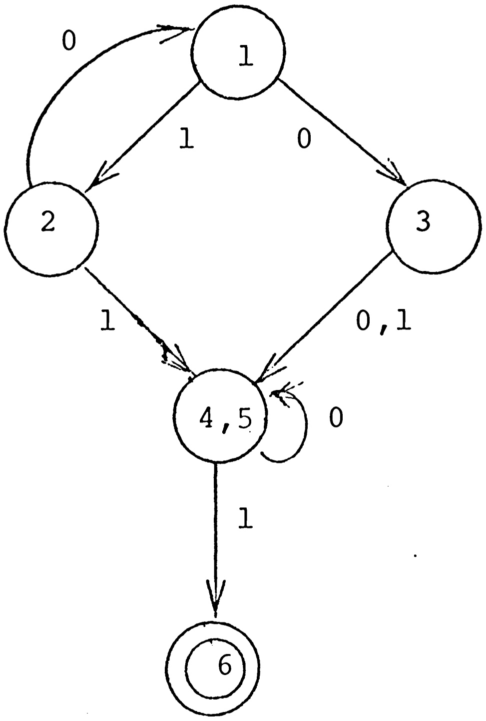
automaton A(S)

For simplicity we can represent the automaton as



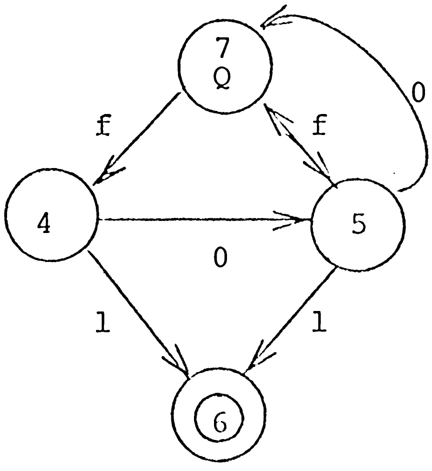
automaton A

When we reduce the automaton A we obtain

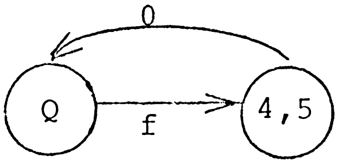


Now it is clear from the reduced automaton that the predicate test in state 3 is redundant, but that state was not removed in the reduction procedure.

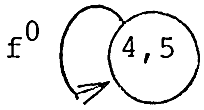
In a more complex situation, the detection of equivalence of states such as 4,5 may depend on knowing that predicates P_i are redundant (redundant perhaps because the states are equivalent). For example, consider adding a state of the following type, labeled 7



Now if 4 and 5 are equivalent, then Q is redundant because it has the form



so one might as well have

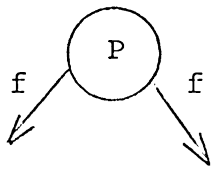


. On the other hand, if Q is not redundant, then 4,5 are not equivalent.

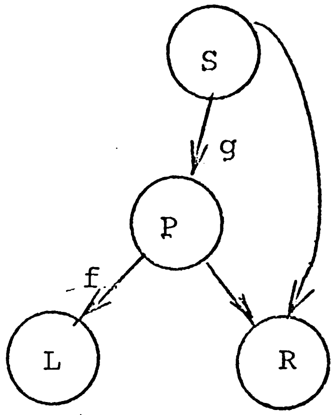
In order to decide whether a predicate test is superfluous we need to apply an algorithm similar to the usual finite automaton reduction technique. We search for nonredundancy. When we find it, we attach the predicate value P_i^+ or P_i^- to the edges leading from the state. Then we repeat the algorithm. Before we describe the technique of detection let us notice that there is a straightforward technique.

<u>Remark</u>: An exponential time test for superfluous predicates.

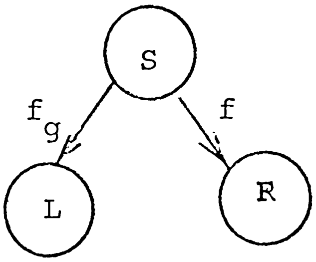
Given an occurrence of a predicate P_i, to test whether it is superfluous we examine whether its careful removal results in an equivalent automaton. That is, given an occurrence of a predicate, it is <u>suspect</u> iff both edges have the same label.

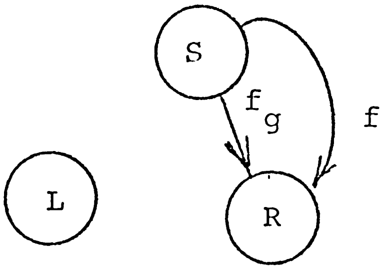


We then remove this state from S and form two new automata S^ℓ, S^r. In S^ℓ we assume that only the left branch need be followed; in S^r we take only the right branch. Thus for instance given the connections



we form





in S^{ℓ} in S^{r}

We then ask whether $S^{\ell} \equiv S^{r}$. To decide this we must call the redundancy test recursively but on smaller automata since P is removed from S^{ℓ} and S^{r}.

We now describe a polynomial time bounded algorithm to reduce the automaton of a strongly free Ianov scheme, A(S).

Informally the algorithm is the usual Moore type reduction algorithm on A(S) except that if a predicate appears to be superfluous at stage n, that is, both branches lead to states which are equivalent at stage n, then it is treated as superfluous (the predicate label is not used in the equivalence algorithm). Whenever a suspected superfluous predicate turns out to be necessary, then we restore the predicate label and recompute the equivalence relation. This algorithm succeeds because if it is possible to reduce A(S) and assume at every stage that a state is redundant, then it is really redundant.

Before we can describe the reduction algorithm we need a number of definitions. First, given an automaton A(S) we associate with each state the predicate P_i of S corresponding to it. Labels from each state have the form xP_i^+, yP_i^- for $x,y \in \{f_j\}^*$. To remove a predicate from a label, say from xP_i^+ or yP_i^-, means to replace these labels by x and y respectively.

In the reduction algorithm we will consider various sets of labels for the edges of the state diagram. At stage n of the algorithm we will use an alphabet denoted $\Sigma^n := \{a_1^n,\ldots,a_{p_n}^n\}$. For any state in the automaton A(S) associated with a strongly free scheme S, at most two of these labels will apply (will lead to a defined transitive or a transition to other than an error state). Call these letters 0_s (the predicate is false) and 1_s (the predicate is true).

As in the Moore type minimization algorithm for finite automata (see [4,5]), we will group states into blocks. The blocks at stage n of the algorithm will be denoted B_i^n.

The algorithm starts with two blocks, $B_1^0 :=$ {halt state}, $B_2^0 :=$ {all non-halt states}, and proceeds to split blocks into smaller blocks until no further splitting is possible. It is possible to split a block B_i^n as long as condition ** given below holds:

$$** \ \exists\, a \in \Sigma^n \ \exists\, s_1,s_2 \in B_i^n \text{ such that}$$
$$\delta(s_1,a) \in B_j^n \quad \delta(s_2,a) \notin B_j^n .$$

That is, there are two states in a block which we can recognize as distinct (inequivalent).

The informal algorithm is this.

Reduction Algorithm

Start with Σ^0 as the set of labels with predicates removed and $A^0(S)$ as the automaton with predicates removed from labels (but written on the states). Let B_1^0 contain the halt state and B_2^0 all non-halt states. Let N be the stage number, initially $N = 0$.

BEGIN

initialize (set N=0, set up B_1^0 , B_2^0).

WHILE ** DO

BEGIN

(1) compute the output behavior of each state under Σ^N (at stage N).

(2) locate the non-redundant states at stage N, i.e.

$$\delta(s,0_s) \in B_i^N \quad \text{and} \quad \delta(s,1_s) \in B_j^N \ , \ i \neq j \ .$$

(3) form a new set of labels, Σ^{N+1}, by restoring the predicates to the labels on the outgoing edges of non-redundant states located in step (2). The new automaton diagram is denoted $A^{N+1}(S)$.

(4) recompute the output behavior using Σ^{N+1}.

(5) split blocks B_i^N to form blocks B^{N+1} by grouping only those states of B_i^N which have the same output behavior as computed in (4).

END

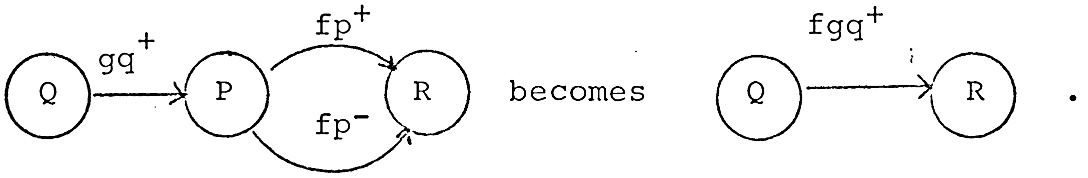
Redundant states are those whose outgoing edges do not have predicates restored to their labels.

END

Given the reduced automaton, say $\hat{A}(S)$, we can construct from it a scheme $\hat{S}$ having no redundant predicates. We remove each redundant state, say

$$L\colon \text{ IF } P_i \text{ THEN } L_\ell \text{ ELSE } L_r$$

and connect all incoming edges to L_ℓ (that is, replace any GOTO L by GOTO L_ℓ). In the state diagram this corresponds to the following



Combining this algorithm with the reduction algorithm we have an algorithm for transforming strongly free Ianov schemes S to reduced strongly free Ianov schemes $\hat{S}$ (we prove this below).

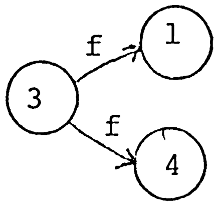
We now examine the details of carrying out steps (1)-(5) in the Reduction Algorithm. We begin with Σ^0, $A^0(S)$.

<u>Details of steps</u>

(1) To compute the output behavior of state s at stage N, find the labels on the outgoing edges of state s; let them be denoted 0_s, 1_s(true). Find the block B_i^N to which the edge labeled 0_s leads and form the pair $<0_s, B_i^N>$. Find the block to which the edge labeled 1_s leads, say B_j^N, and form the pair $<1_s, B_j^N>$. Now form a finite function from Σ^N to $\{B_i^N\} \cup \{\Omega\}$ by adding the pairs

$<a_i^N,\Omega>$ for $a_i^N \neq 0_s$, $a_i^N \neq 1_s$. Let F_s^N denote this function, so $F_s^N\colon \Sigma^N \to \{B_i^N\} \cup \{\Omega\}$

Notice that when the predicates are removed from labels, it appears that the state transitions are non-deterministic,

e.g. 

may appear. But in fact either the target states, B_i^N and B_j^N are distinct $(i \neq j)$ in which case in step (4) we restore the labels and remove the non-determinism or else they are identical $(i=j)$ in which case the function value is uniquely determined and hence deterministic.

(2) To locate the non-redundant states s at stage N, search the states whose predicates have not yet been restored (in step (3)) and find those such that either:

(i) $0_s \neq 1_s$ or
(ii) $0_s = 1_s$ and $B_i^N \neq B_j^N$ in notation of step 1
(hence F_s is non-deterministic)

Restore the predicates to the labels on the outgoing edges of those states. Call the new state diagram $A^{N+1}(S)$.

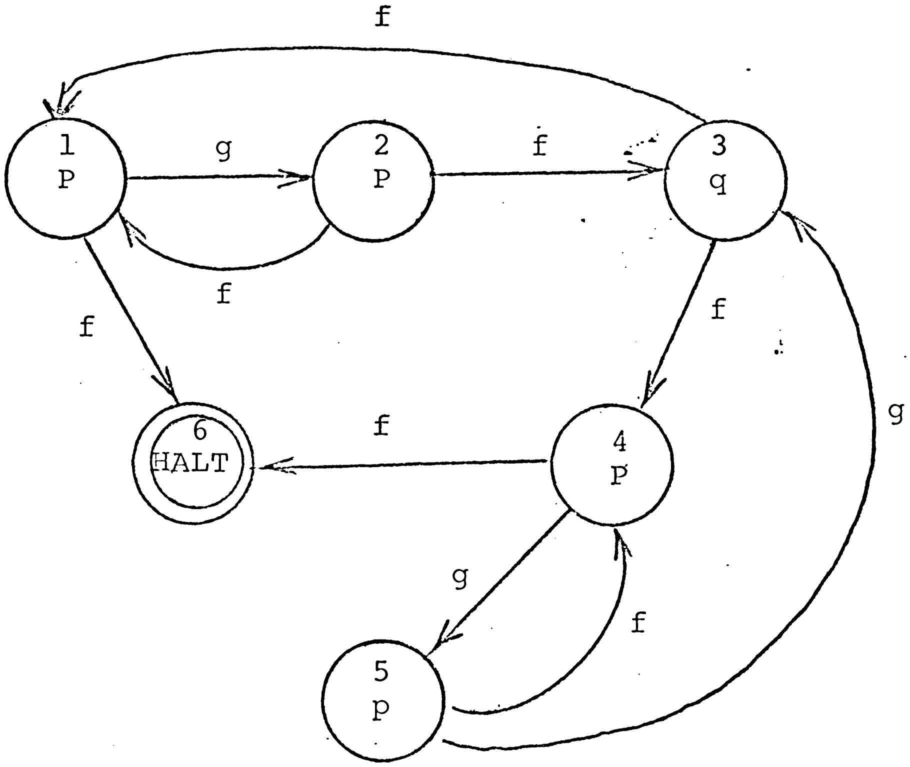
(3) To form a new set of labels, Σ^{N+1}, take the labels of edges in $A^{N+1}(S)$.

(4) To recompute the finite functions F_s^N, proceed as in step (1) using the labels Σ^{N+1}. Notice that all such functions, denoted $\bar{F}_s$, are deterministic.

(5) Split each block B_i^N into sub-blocks of states having identical output behavior, i.e. $\bar{F}_s = \bar{F}_t$. Let the new blocks be numbered and denoted $B_1^{N+1}, \ldots, B_p^{N+1}$.

This completes a detailed description of the algorithm. We next consider an example, then we analyze the running time to obtain a crude but sufficient upper bound $(O(|\Sigma| \cdot |K|^2))$. Finally we prove the correctness in Theorem 3.2.

Example 3.7:



$A^0(S)$

B_1^0	B_2^0
{6}	{1,2,3,4,5}

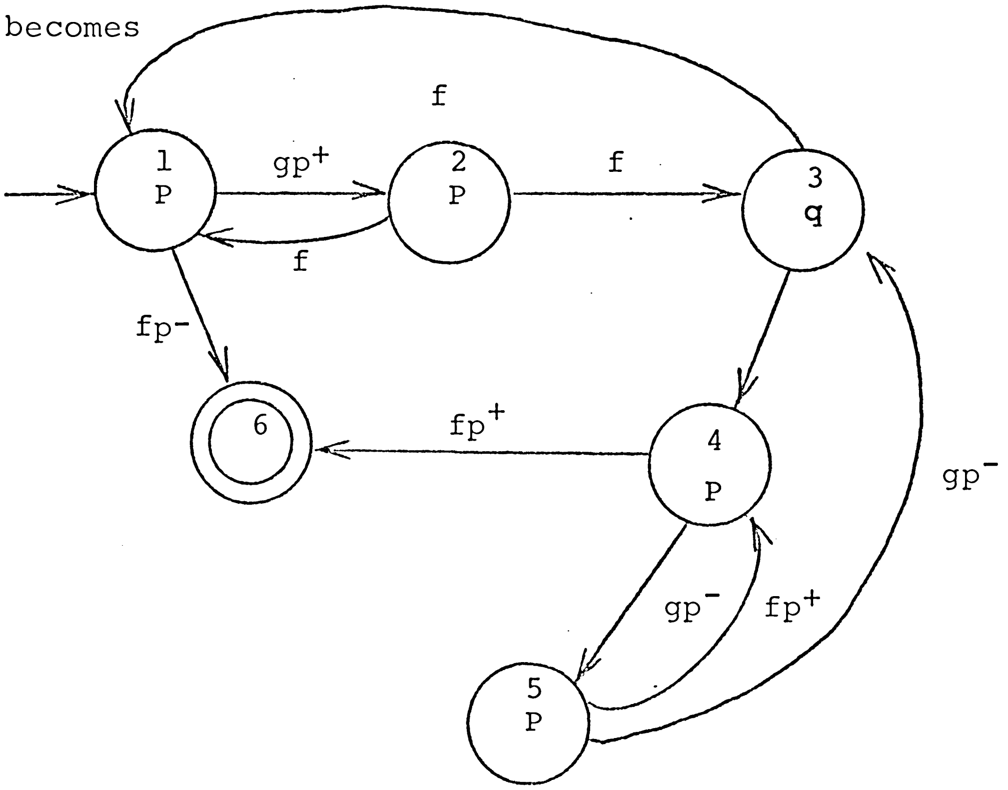
To understand the relationship of this algorithm to the Moore minimization algorithm, consider the following temporary grouping which is not part of the algorithm.

F_1	F_2	F_3	F_4	F_5
$\langle f, B_1^0 \rangle$	$\langle f, B_2^0 \rangle$	$\langle f, B_2^0 \rangle$	$\langle f, B_1^0 \rangle$	$\langle f, B_2^0 \rangle$
$\langle f, B_2^0 \rangle$	$\langle g, \Omega \rangle$	$\langle g, \Omega \rangle$	$\langle g, B_2^0 \rangle$	$\langle g, B_2^0 \rangle$
$\langle g, \Omega \rangle$				

A suggestive grouping is

{6}   {1,4}   {2,3}   {5}

but this is not the B' grouping because Σ' becomes $\{fp^-, gp^+, fp^+, gp^-, f\}$ since states 1,4,5 are recognized to be non-redundant. Then A'(S) becomes



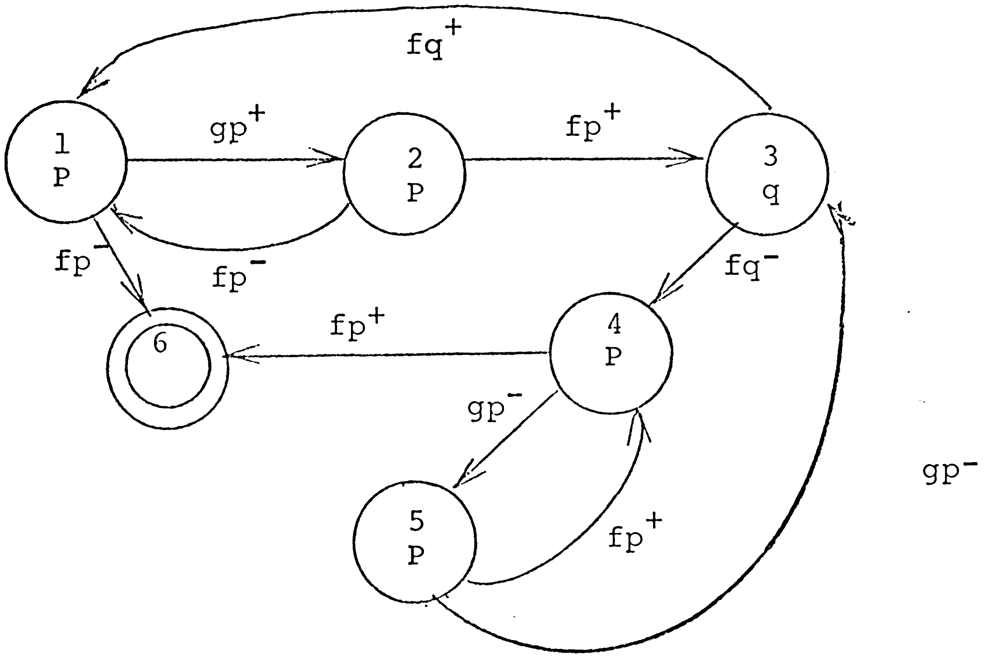
So the finite functions are

$\overline{F}_1$	$\overline{F}_2$	$\overline{F}_3$	$\overline{F}_4$	$\overline{F}_5$
f,Ω	B_2^0	B_2^0	Ω	Ω
fp^+,B_2^0	Ω	Ω	B_1^0	B_2^0
fp^-,B_1^0	Ω	Ω	Ω	Ω
gp^+,Ω	Ω	Ω	Ω	Ω
gp^-,Ω	Ω	Ω	B_2^0	B_2^0

The B_i^1 blocks are for $i = 1,2,3,4,5$:

B_1^1	B_2^1	B_3^1	B_4^1	B_5^1
$\{6\}$	$\{1\}$	$\{2,3\}$	$\{4\}$	$\{5\}$

Finally $A^2(S)$ is



and we see that no states are redundant.

The reader should try the algorithm on scheme S_1 in Example 3.4 item (2) and verify that S_2 is the minimal scheme.

Analysis of runtime

It is easy to see that the Reduction Algorithm is in the worst case bounded above by $O(|\Sigma|\cdot|K|^2)$. Consider the time for each step, the bounds are

(1) $\leq |\Sigma|\cdot|K|$ (2) $\leq |K|$ (3) $\leq |\Sigma|$ (4) $\leq |\Sigma|\cdot|K|$ (5) $\leq |K|$

So the worst case occurs when at most one state is split off of a block on each iteration. Thus the worst case is

$$|K|\cdot(|\Sigma|\cdot|K| + |K| + |\Sigma|\cdot|K| + |K|) \leq 5\cdot|\Sigma|\cdot|K|^2.$$

If we use a more efficient algorithm, such as Hopcroft [5] (also see Gries [4]), then the time is $O(|\Sigma|\cdot|K|\cdot\ell og(|K|))$. In any case this is a polynomial time bounded algorithm in either $|K|,|\Sigma|$ or in $|S|$.

We now summarize our knowledge in a theorem.

Correctness of the algorithms.

Theorem 3.2: There is an algorithm whose runtime is no more than a polynomial in $|S|$ which produces the reduced scheme $\hat{S}$ given S. That is,

(i) $S \equiv \hat{S}$ and

(ii) S contains no redundant predicates.

Proof:

The time analysis given above shows that the algorithm is polynomial in $|S|$. We need only show (i) and (ii). We first consider (i).

(1) Clearly if a predicate P_i remains in S then it is not redundant because the algorithm produces an interpretation under which the true and false branches from P_i are distinct. So we need only show that if a predicate occurrence is removed, say at state s as

$$s\text{: IF } P_i \text{ THEN } L_\ell \text{ ELSE } L_f ,$$

then that occurrence is really redundant. To prove this, suppose some predicates occurrences were erroneously removed, say P_{i_1} at state $s_{i_1}, \ldots, P_{i_n}$ at state s_{i_n}. Then order these by the length of the interpretation under which the true and false branches are distinct. Suppose P_i is one with the least such length interpretation. Then that interpretation cannot involve another predicate erroneously removed in an essential way, that is either the two computations, the true one which is $x_n x_{n-1} \ldots x_1$ or the false one, $y_m y_{m-1} \ldots y_1$ either (a) do not contain any P_{i_j} (erroneously removed predicates) or (b) if such a P_{i_j} does occur, then the true branch from it to the halt state (x_n or y_m) is the same as the false branch, because otherwise this P_{i_j} would have a shorter interpretation showing it to be non-redundant than P_i does, contradicting the definition of P_i. Thus in either case (a) or (b), the computations $y_m \ldots y_1$ and $x_n \ldots x_1$ appear already in some $A^k(S)$. That is, neither computation requires the presence of an erroneously classified predicate. Therefore, P_i would be discovered to be non-redundant at some stage k of the Reduction Algorithm.

(2) Finally, to show $S \equiv \hat{S}$ we notice that S and $\hat{S}$ are nearly isomorphic. For every state s of S there is a corresponding state $\hat{s}$ of $\hat{S}$ unless s is redundant. But if s is redundant, then

we know that the edges in S which by-pass s do not change equivalence. The reader can prove this carefully by considering these "near isomorphisms" under any Herbrand interpretation H.

Q.E.D.

We now state a fact about finite automata.

Theorem 3.3: There is an $O(|\Sigma| \cdot n \ell og(n))$ time algorithm to decide the equivalence of finite automata S_1, S_2 over Σ where $n = \max(|K_1|, |K_2|)$, K_1, K_2 the state sets of S_1, S_2.

Using this we have the theorem we need.

Theorem 3.4: There is a polynomial time bounded algorithm to decide the equivalence of strongly free Ianov schemes.

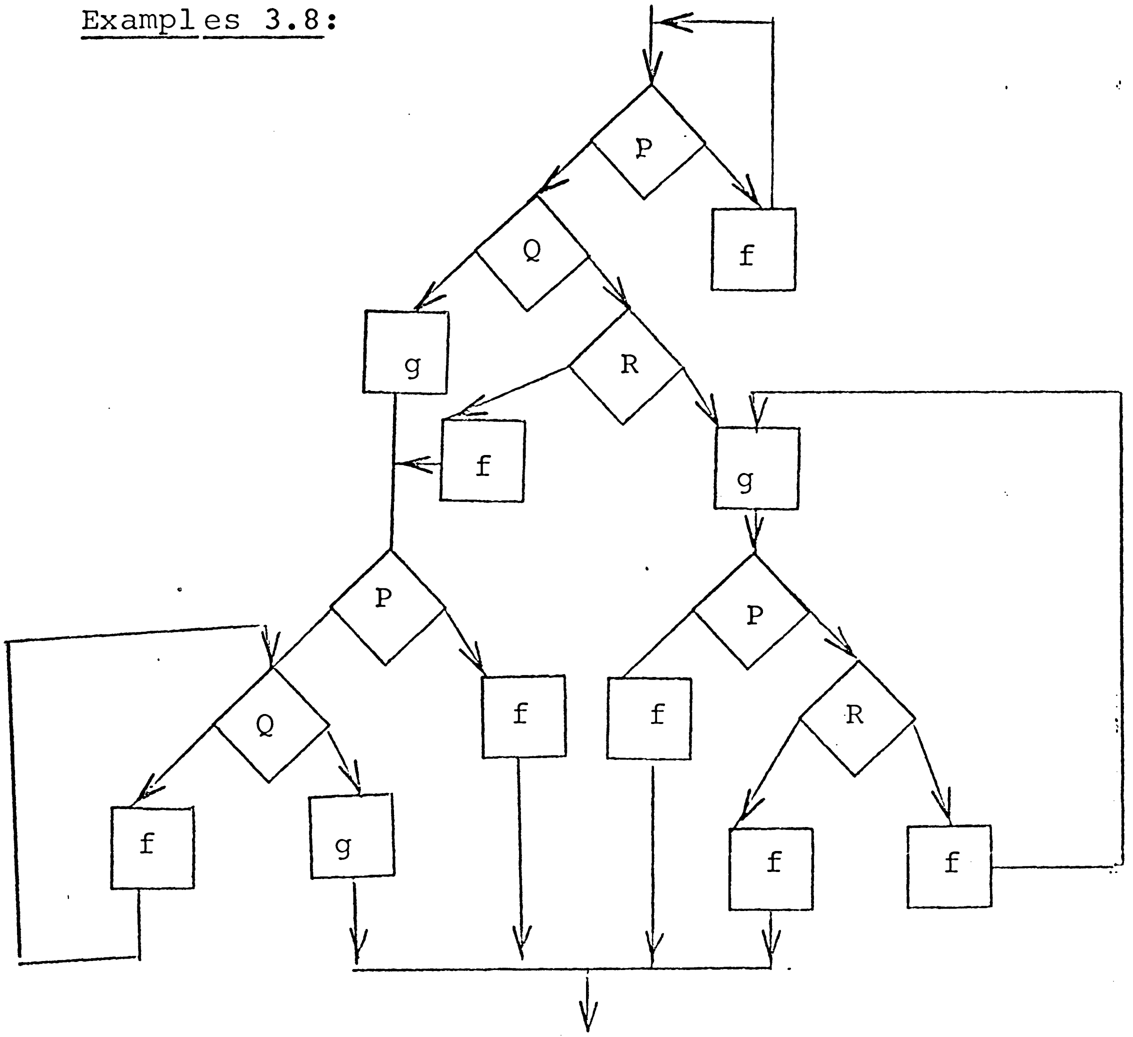
Proof: Apply Theorem 3.1, 3.2, 3.3.

Extension to free Ianov schemes

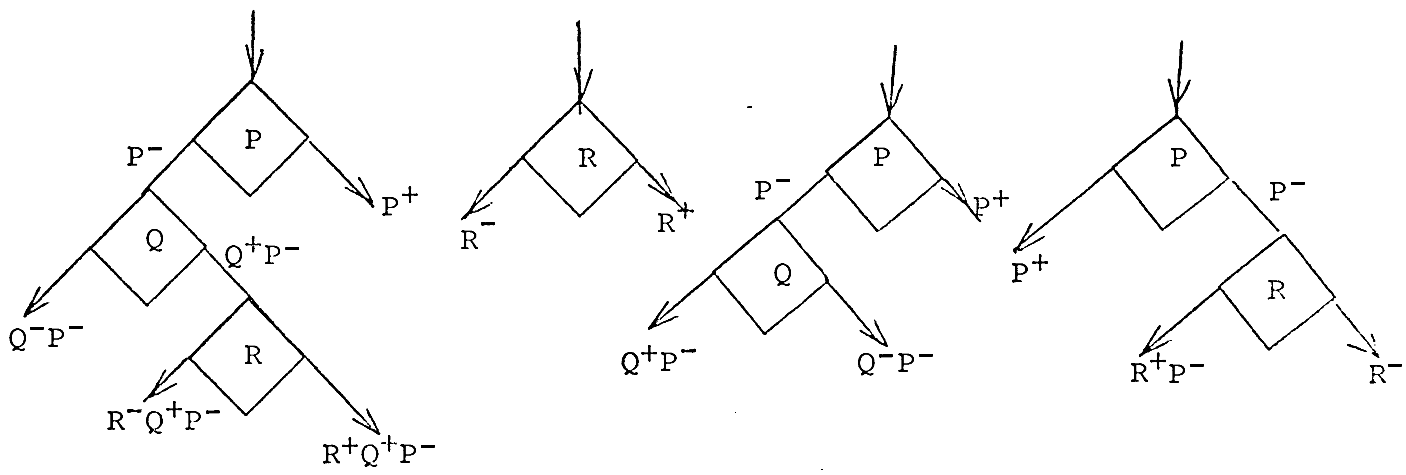
We now want to extend the reduction and equivalence algorithms from strongly free Ianov schemes to strongly free Ianov schemes with tree-like predicate clusters substituted for predicates. The idea is to replace any tree-like cluster of predicates by a single multi-exit predicate.

Definition 3.5: Let S be a Ianov scheme, then a cluster of predicates in S is a loop free subscheme of S containing no function applications and such that no edge can be extended without including a function application. A tree-like cluster is such that the cluster is a tree whose nodes are predicates.

Examples 3.8:

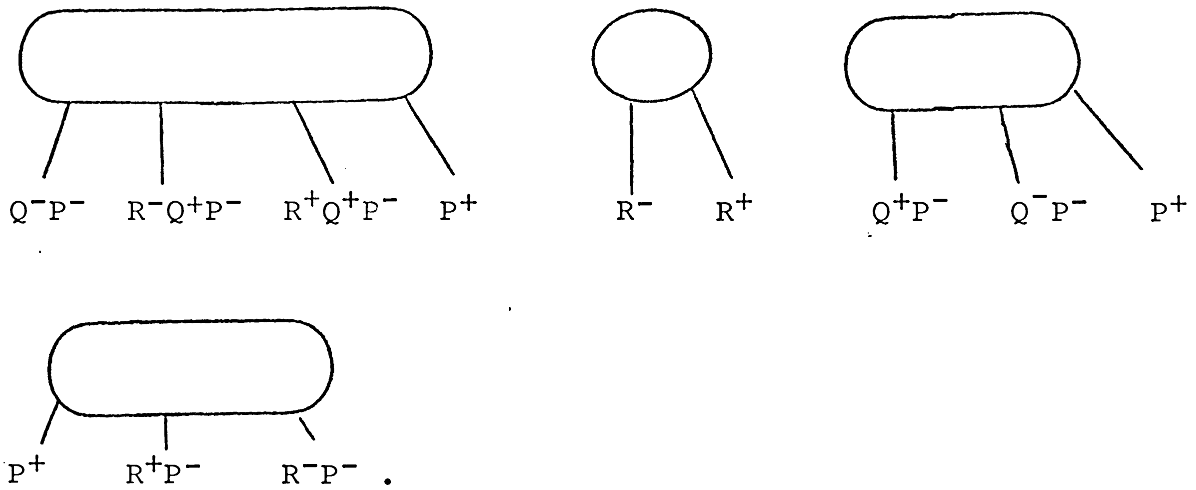


The tree-like clusters are

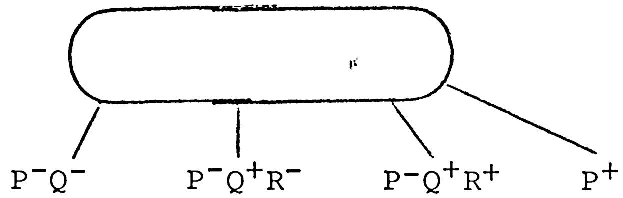


Notice that in a free scheme no predicate can occur more than once on a path from root to leaf in a cluster, but predicates may indeed occur more than once.

We represent these clusters by multi-exit predicates:

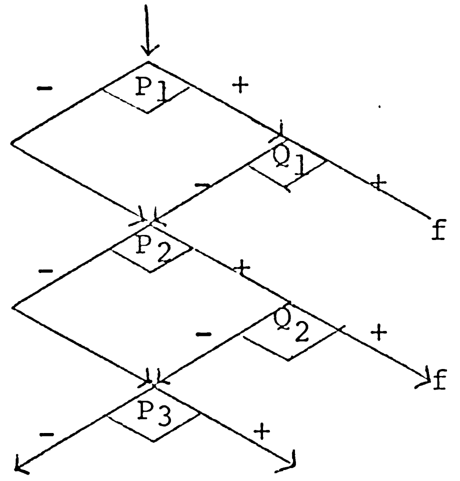


We can make this assignment of multi-exit predicates to clusters uniform if we choose a specified ordering of predicates. Say we have P, Q, R, T. We then agree to label all outputs in the order P, Q, R, T. Thus the first cluster in the example becomes

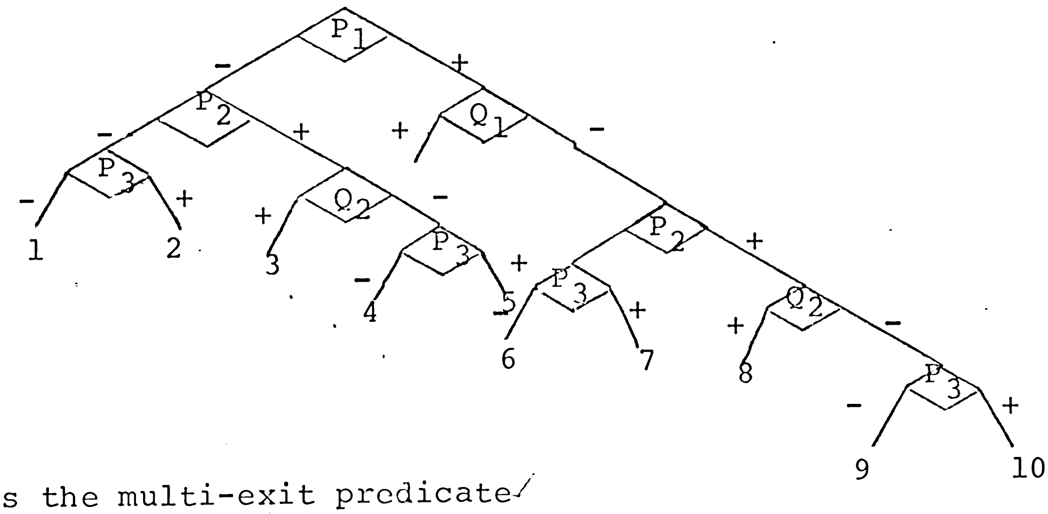


To decide equivalence of free Ianov schemes S_1, S_2 we convert the predicate clusters to multi-exist predicates and then convert the result to a finite automaton, A(S), with labels on predicates given in a standard order. In the case of free Ianov schemes, the generation of multi-exit predicates may require exponential time. Consider the following example.

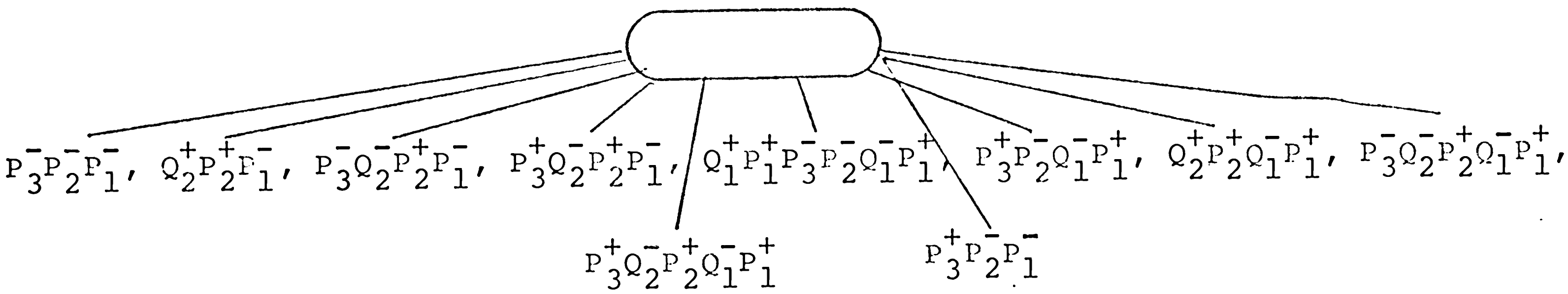
Example 3.9:



This cluster generates the tree-like cluster



which generates the multi-exit predicate



If all the predicate clusters in a Ianov scheme are tree-like, then the multi-exit predicate has the same number of exits as there are leaves in the tree, thus it can be generated in polynomial time (in the number of edges) given the cluster. We state this as a proposition.

Proposition 3.4: There is an algorithm which accepts a tree-like predicate cluster with n leaves and generates in time cn a multi-exit predicate with labels as described above.

The main difficulty in carrying over the results for strongly free schemes is formulating the reduction algorithm. Once this is done we use the same type of theorem as before. Namely,

Theorem 3.5: If S_1, S_2 are free Ianov schemes then

$$S_1 \equiv S_2 \iff \hat{A}(S_1) \equiv \hat{A}(S_2)$$

where $\hat{A}(S_i)$ is a finite automaton with no superfluous edges.

Thus to prove our main result we only need

Theorem3.6: There is a polynomial time bounded algorithm to reduce any free Ianov scheme.whose only predicate clusters are tree-like.

Then applying Theorems 3.5, 3.6 we have

Theorem3.7: There is a polynomial time bounded algorithm to decide the equivalence of free Ianov schemes.

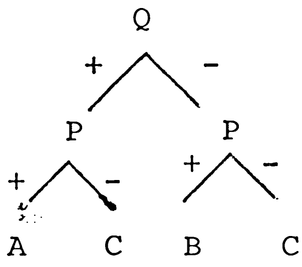
So to finish this section we need only prove Theorem 3.6. We use the same type of reduction algorithm as before but we must be careful to say exactly when a predicate in a cluster is redundant on the basis of information gathered about the multi-exit predicate in A(S).

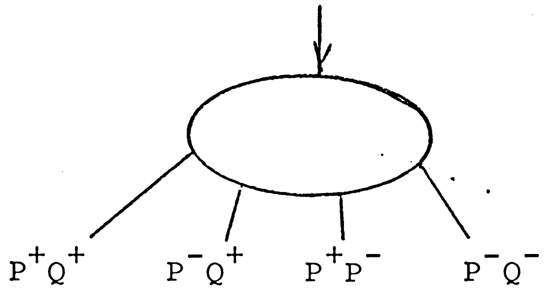
During the reduction algorithm, the edges leaving a multi-exit predicate will be grouped together into edge-groups, $E_i^N(s)$; that is at stage N there may be $i = 1,\ldots,m$ edge-groups associated with state s.

Definition 3.6 : We say that a predicate occurrence Q in a cluster C is redundant with respect to the edge-groups E_i^N iff for all sequences of predicate tests z_1 such that $z_1 Q^+ y \in E_i^N$ there is a sequence z_2 compatible with z_1 (no predicate P appears as P^+ in z_1

and P^- in z_2 or vise versa) such that $z_2Q^-y \in E_i^N$. That is, Q does not affect the decisions made by predicates tested after Q.

Example 3.10: Let the edge-groups be labeled A,B,C.





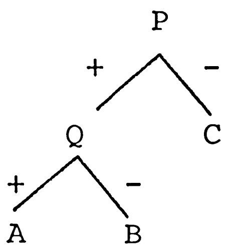
cluster

multi-exit predicate

The sequences in the edge-groups are

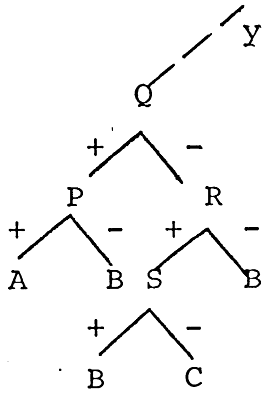
A	B	C
P^+Q^+	P^+Q^-	P^-Q^+
		P^-Q^-

We see that predicate Q is redundant with respect to edge-group C. (Therefore in the reduction algorithm we will replace the labels P^-Q^+, P^-Q^- by P^-.) One can see the redundancy more clearly if the predicate cluster is rewritten as



Remark: This example suggests how inefficient a predicate cluster might be. We do not need to consider methods of finding the minimum cluster equivalent to a given cluster in order to obtain a polynomial equivalence algorithm. We only need a method of eliminating the redundant predicates from the labels on outgoing edges of multi-exit predicates.

This example's too simple to illustrate the difficulties in testing for redundant predicate occurrences. It is not sufficient to see whether xQ^+y and xQ^-y both appear in an edge group. Consider this example.



Then in edge-group B we have P^-Q^+y, $S^-R^+Q^-y$, R^-Q^-y. So Q is redundant for B because both S^-R^+ and R^- are compatible with P^-.

In order to mimic the reduction algorithm for strongly free schemes, we need a procedure to check for redundancy in predicate clusters given an assignment of edge-groups (this assignment comes from the main algorithm).

Multi-exit non-redundancy procedure.

Given predicate cluster C and edge-groups $E_1,\ldots,E_m$, to test whether a predicate occurrence Q in a label on an edge in E_i^N is redundant, do the following:

BEGIN

(1) locate Q in the cluster (let y be the path to Q).

(2) list all prefixed of the form z where $zQ^{+}y$ is in E_i^N.

(3) for each z in (2) check whether there is a prefix w where (a) $wQ^{-}y$ is in E_i^N

(b) w and z are compatible.

(4) if there is a w for each z, then Q is redundant, otherwise it is not and the predicate is output.

END

The reader can easily check the correctness of this procedure.

Proposition 3.5: A predicate occurrence Q in tree-like cluster C is non-redundant wrt edge group E_i iff the multi-exit redundancy procedure generates Q given C and E_i.

It is also easy to check that this procedure runs in polynomial time in the number of predicates in the cluster.

Proposition 3.6: If tree-like predicate cluster C has n predicates, then the multi-exit redundancy procedure runs in at most n^2 steps.

§4 Simple Programs

In this section we show that deciding equivalence of programs from certain very elementary languages is NP-Complete. We begin by looking at the well-known loop languages and then we look at other simple languages.

Definition 4.1: A Loop program is a finite sequence of instructions of five types:

a) DO X

b) END

c) $X \leftarrow 0$

d) $X \leftarrow Y$

e) $X \leftarrow X + 1$

Definition 4.2: $\mathcal{L}_i$ is the set of Loop programs in which the maximum level of resting of the DO's is i. Inequiv ($\mathcal{L}_i$) is the problem of deciding whether two $\mathcal{L}_i$ programs are inequivalent.

Theorem 4.1: Inequiv ($\mathcal{L}_1$) is NP-Complete.

Proof: a) Inequiv ($\mathcal{L}_1$) is NP-hard.

We show how, for each formula P, to construct a program, P_1, in $\mathcal{L}_1$ such that Output$(P_1) = 0$ for all inputs, $(x_1,\ldots,x_n)$, iff the formula P is not satisfiable.

$$\text{Let } P = \bigwedge_{i=1}^{k} C_i$$

$$\text{and } C_i = \bigvee_{j=1}^{k_i} C_{ij}$$

where each C_{ij} is a literal.

Let the variables of P be $x_1, x_2, \ldots, x_n$.

The $\mathscr{L}_1$ program P_1 is then constructed as below:

P_1 has n input variables $x_1,\ldots,x_n$.

$x_i = 0$ will correspond to a truth value of 'False' for variable x_i of formula P.

$x_i > 0$ will correspond to a truth value of 'True' for variable x_i of formula P.

We define the following program blocks:

1) A_i

This computes $\bar{x}_i$ which corresponds to the complement of x_i

$$A_i = \left[\begin{array}{l} \bar{x}_i = 1 \\ \text{DO } x_i \\ \bar{x}_i = 0 \\ \text{END} \end{array}\right.$$

$$\text{Thus } \bar{x}_i = \begin{cases} 1 & \text{if } x_i = 0 \\ 0 & \text{otherwise} \end{cases}$$

2) B_i

For any given values of $x_1,\ldots,x_n,\bar{x}_1,\ldots,\bar{x}_n$, this computes, C_i , the value of the ith clause of formula P.

We illustrate the construction of B_i by an example. Suppose the ith clause is $x_1 \vee \bar{x}_2 \vee x_3$ then we have

$$B_i = \left[\begin{array}{l} C_i = 0 \\ \text{DO } X_1 \\ C_i = 1 \\ \text{END} \\ \text{DO } \bar{X}_2 \\ C_i = 1 \\ \text{END} \\ \text{DO } X_3 \\ C_i = 1 \\ \text{END} \end{array}\right. \qquad \text{Thus } C_i = \begin{cases} 1 & \text{if the ith clause is true} \\ 0 & \text{otherwise} \end{cases}$$

3) $\underline{D_i}$

This code computes $\bar{C}_i = \begin{cases} 1 \text{ if } C_i = 0 \\ 0 \text{ otherwise} \end{cases}$ $\qquad D_i = \left[\begin{array}{l} \bar{C}_i = 1 \\ \text{DO } C_i \\ \bar{C}_i = 0 \\ \text{END} \end{array}\right.$

The program P_1 then has the form:

Program P_1:

INPUT $(X_1,X_2,\ldots,X_n)$; A_1; A_2; ...; A_n; B_1; ...; B_k; D_1; ...; D_k; P = 1; DO $\bar{C}_1$; P = 0; END; ...; DO $\bar{C}_k$; P = 0; END; OUTPUT(P)

Clearly, program P_1 outputs a 1 for some input $(x_1,\ldots,x_n)$ iff the formula P is sastisfiable. Otherwise it always outputs '0'.

Hence $P_1 \equiv 0$ iff P is not satisfiable (0 is the trivial loop program $\left[\begin{array}{l} x=0 \\ \text{Output}(x) \end{array}\right]$)

b) Inequiv($\mathcal{L}_1$) $\in$ NP)

This follows immediately from theorems 4 and 7 of Tsichritzis [12].

<u>Remark 4.1</u>: The Equivalence problem for $\mathcal{L}_0$ is solvable in linear time as each program P_i in $\mathcal{L}_0$ defines a function f_i of the type $x_j + k_i$ or k_i. The particular f_i being computed can be determined

in polynomial time. (See Tsichritzis [12]).

Remark 4.2: The equivalence problem for $\mathscr{L}_2$ is undecidable (See Meyer and Ritchie [14]).

Definition 4.3: P_1 is the class of programs defined by all finite sequences of statements of type

a) $x_i = x_j + x_k$ and $x_i = 1 \dot{-} x_j$ (the x_j and x_k may be constants).

P_2 is the class of programs defined by all finite sequences of statements of type $x_i = x_j \dot{-} x_k$ (the x_j and x_k may be constants).

Theorem 4.2: Equiv(P_1) is NP-Complete.

Proof: Satisfiability $\leq_{\text{ptime}}$ Inequiv(P_1)

For any formula P in conjunctive normal form we construct a program $P_1 \in P_1$ such that Output$(P_1) \equiv 0$ iff P is not satisfiable.

Let $P = \bigwedge_{i=1}^{k} C_i$

$$C_i = \bigvee_{j=1}^{m_i} C_{ij}$$

and variables of $P = x_1, x_2, \ldots, x_n$.

The program P_1 has n input variables $x_1, \ldots, x_n$ where $x_i = 0$ will correspond to a truth value of False for the variable x_i of P. All other values of x_i would correspond to a truth value True (note that the inputs to program P_1 are always nonnegative).

For each clause C_i we define a program segment D_i that computes its "value", i.e. $C_i = 0$ iff clause C_i is false. The construction of D_i is similar to that for loop and so will not be given.

Remark 4.3: For programs using only instructions of type $x_i = x_j + x_k$ the equivalence problem takes only linear time. Now, we have

$$\text{Output}(P_1) = \sum_{k=1}^{\ell} c_k \, x_{i_k} + C_0$$

$$\text{Output}(P_2) = \sum_{k=1}^{m} c_k \, x \, j_k + \hat{C}_0$$

where the c_i are constants and x_i's are variables and $P_1 \equiv P_2$ iff the expressions for output(P_1) and Output(P_2) are identical. The output expressions can be obtained in linear time on a random access machine.

Remark:4.4: The equivalence problem for programs using only statements of the type $x_i = 1 \dot{-} x_j$ is polynomially solvable. It is easy to see that the output expressions for P_1 and P_2 are of the type:

$$\text{Output}(P_1) = (1 \dot{-} (1 \dot{-} \ldots\ldots \dot{-} x_i)\ldots)$$
$$\text{Output}(P_2) = (1 - (1 \dot{-} \ldots\ldots \dot{-} x_j)\ldots)$$

and $P_1 \equiv P_2$ iff the x_i and x_j are the same variables or constant and also the number of '$\dot{-}$' signs in both expressions is either odd or even or zero.

The output expressions are easily obtained by starting from the output statement in each program and back substituting until one reaches the input statement.

Remark 4.5: The programs of remark 3.4 can be extended to include finite sequences of statements of type $x_i = k_\ell - x_j$. The equivalence problem is still decidable in polynomial time. To see this, we note that the output functions as obtained using the procedure suggested in remark 3.4 are of the form:

$$\text{Output}(P_1) = (k_1 \dot{-} (k_2 \dot{-} \ldots\ldots \dot{-} x_i)\ldots)$$

and $$\text{Output}(P_2) = (\hat{k}_1 - (\hat{k}_2 - \ldots - x_j)\ldots)$$

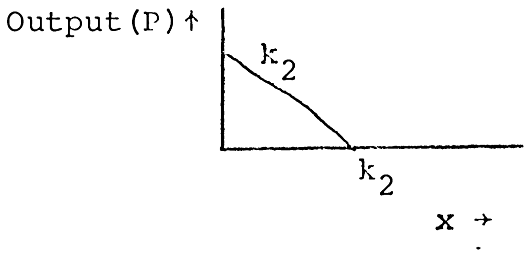
Now, $P_1 \equiv P_2$ iff (1) x_i and x_j are the same variable or constant

and (2) the range graphs for P_1 and P_2 are identical for all integral values of x_i ($x_i \geq 0$).

Definition 4.4: The range graph of a program P is a plot of Output(P) as a function of input values (The input is restricted to positive real numbers).

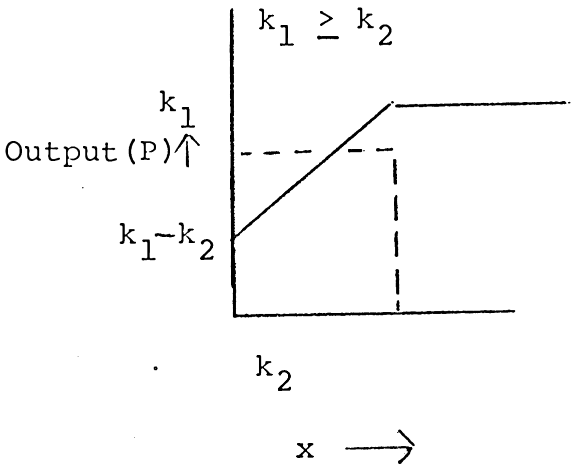
For programs with output functions of the type (*) range graphs are obtained as below.

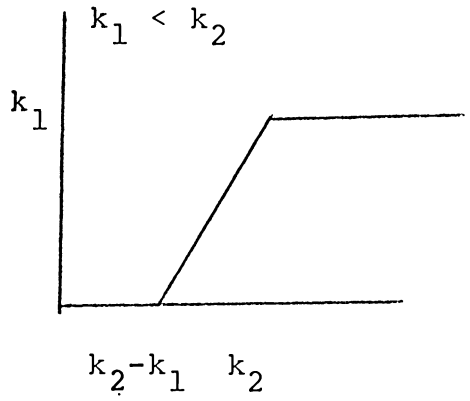
a) If Output(P) $= k_2 \dot{-} x$ then we get



The output is completely characterized by the two set of coordinates (C_2, k_2) and $(k_2, 0)$

b) Output(P) $= k_1 \dot{-} (k_2 \dot{-} x)$





For $k_1 \geq k_2$ the output is characterized by the coordinate pairs $(0, k_1 - k_2)$, (k_2, k_1). For $k_1 < k_2$ the relevant pairs are $(0,0)$, $(0, k_2 - k_1)$ and (k_2, k_1).

For an expression of type $(k_1 \dot{-} (k_2 \dot{-} \ldots \dot{-} (k_\ell \dot{-} x)\ldots)$ at most $\ell+1$ pairs of coordinates are needed to characterize its range graph. Given the k_i's , these points can be easily computed in polynomial time using elementary trignometry.

[A] Output(P_1) =
$$\begin{cases} m_1^1 x + c_1^1 & \leq x \leq b_1^1 \\ \vdots & \\ m_k x + c_k^1 & a_k^1 \leq x \leq b_k^1 = \alpha \end{cases}$$

[B] Output(P_2) =
$$\begin{cases} m_1^2 x + c_1^2 & 0 \leq x \leq b_1^2 \\ \vdots & \\ m_\ell^2 x + c^2 & a_\ell^2 \leq x \leq b_\ell^2 \end{cases}$$

To test for integer equivalence of range graphs we need

1) If a line segment is valid for only one integer value of x then the outputs of P_1 and P_2 (as given by the relevant line segments) are computed and checked for equality.

2) If a line segment ℓ_i is valid for P_1 and for a line segment ℓ_j valid for P_2 has at least two integer values of x in common with ℓ_i then for the outputs to be identical $m_i^1 = m_j^2$ and $c_i^1 = c_j^2$.

So, using the line segments A[A] and [B] it is possible to check for integer equivalence in polynomial time.

What happens if we try to extend the programs of remarks 3.4 and 3.5 further. Suppose the k_i are replaced by variables i.e. the programs now are finite sequences of statements of the type $x_i = x_j \dot{-} x_k$ where the x_j and x_1 may be constants. This is the class P_2 of definition 3.3. The equivalence problem now becomes NP-Complete!

Theorem 4.3: Equiv($\mathscr{P}_2$) is P-Complete.

Proof: (a) Satisfiability $\leq_{ptime}$ Equiv ($\mathscr{P}_2$)

We construct a $_2$ program, P_1, whose size is linear in the size of the formula P such that Output$(P_1) \equiv 0$ iff P is not satisfiable. Once again, we let $P = \bigwedge_{i=1}^{k} c_i$, $c_i = \bigvee_{j=1}^{\ell_i} c_{i_j}$ and $\mathrm{Var}(P) = x_1, x_2, \ldots, x_n$.

The program uses program blocks of type D. Each such block D_i computes a value C_i such that $C_i = 0$ if the clause C_i is false for the given truth values of the x_i's. Each such D block assumes that the x_i's are 0/1 valued.

For example if $C_i = x_1 + \bar{x}_2 + x_3$ then

$$D_1 = \begin{array}{l} C_1 = 3 \dot{-} \bar{x}_1 \\ C_1 = C_1 \dot{-} x_2 \\ C_1 = C_1 \dot{-} \bar{x}_3 \end{array}$$

So $C_1 = 0$ iff all the terms in clause C_1 are false (ie zero).

Program P_1

$$\begin{array}{ll}
\text{INPUT}(x_1, x_2, \ldots, x_n) & \\
\left.\begin{array}{l} \bar{x}_1 = 1 \dot{-} x_1 \\ x_1 = 1 - x_1 \\ x_n = 1 \dot{-} x_n \\ x_n = 1 \dot{-} \bar{x}_n \end{array}\right\} & \text{set } x_i, \bar{x}_i = 0/1 \\
D_1 & \\
\vdots & \\
D_k & \\
\left.\begin{array}{l} C_1 = 1 \dot{-} C_1 \\ C_1 = 1 \dot{-} C_1 \\ \quad\vdots \\ C_k = 1 \dot{-} C_k \\ C_k = 1 \dot{-} C_k \end{array}\right\} & \begin{array}{l} C_i = 0 \text{ if clause } C_i \text{ is false} \\ \quad = 1 \text{ otherwise.} \end{array} \\
\left.\begin{array}{l} P = k \dot{-} C_1 \\ P = P \dot{-} C_2 \\ \quad\vdots \\ P = P \dot{-} C_k \\ P = 1 \dot{-} P \end{array}\right\} & \begin{array}{l} \text{Compute } P \text{ such that} \\ \qquad P = 0 \text{ if at least one clause} \\ \qquad\qquad \text{is false} \end{array} \\
\text{Output}(P) &
\end{array}$$

From the construction of P_1 it follows that Output$(P_1) \equiv 0$ iff P is not satisfiable. Further, the construction of P_1 takes only polynomial time.

b) Inequiv$(\mathscr{P}_2)$ is in NP.

§5 Conclusion

Open Problems

P1: How hard are divergence, halting, and freedom for recursion schemas?

P2: What about containment and weak equivalence for single variable free schemes? Are they in P?

P3: How hard is equivalence for free loop-free schemas? Is it in P?

P4: Are there other interesting subclasses of free schemas, membership in which is P-decidable?

predicate	Ianov tree	tree	loop-free Ianov [1]	free Ianov	free	2-variable loop-free [1]	linear monadic recursion
divergence	P	P	NP-complete	P	P	NP-complete	NP-complete
halting	P	P	NP-complete	P	P	NP-complete	NP-hard
freedom	P	P	P	T	T	NP-complete	?
strong equivalence	P	?	NP-complete	?	?	NP-complete	NP-complete
weak equivalence	?	?	NP-complete	?	?	NP-complete	NP-hard
containment	P	P	NP-complete	?	?	NP-complete	NP-hard
isomorphism	P	P	NP-complete	P	P	NP-complete	

1. Allowing LOOP instructions

T means "trivial"

P means "in P"

? means that the complexity is unknown

References

[1] Ashcroft, E., Z. Manna, and A. Pneuli, "Decidable Properties of Monadic Functional Schemes", JACM, 20, 3, July 1973, 489-499.

[2] Cook, S.A., "The complexity of theorem proving procedure", Proc. 3rd Annual ACM Symposium on Theory of Computing.

[3] Garland, S.J., and D.C. Luckham, "Program Schemes, Recursion Schemes and Formal Languages", JCSS 7, 119-160.

[4] Gries, D., "Describing an Algorithm by Hopcroft", Acta Informatica 2, 1973, 97-109.

[5] Hopcroft, J.E., "An n ℓog n Algorithm for Minimizing States in a Finite Automaton", Theory of Machines and Computations, ed. Z. Kohavi & A. Paz, Academic Press, 1971, 189-196.

[6] Hopcroft, J.E., and J.D. Ullman, Formal Languages and their Relation to Automata, Addison-Wesley, Reading, 1969, 242.

[7] Manna, Z., "Program Schemas" in Currents in the Theory of Computing, A.V. Aho, editor, Prentice-Hall, Englewood Cliffs, 1973, 90-142.

[8] Luckham, D.C., D.M.R. Park, and M.S. Paterson, "On Formalized Computer Programs", JCSS 4,3, 220-249.

[9] Kaplan, D.M., "Regular expressions and the equivalence of programs", JCSS 4,3, 361-386.

[10] Routledge, J.D., "On Ianov's Program Schemata", JACM, 11, 1, Jan. 1964, 1-9.

[11] Ianov, I., "On Logical Algorithm Schemata", Cybernetics Problems, I, Moscow, 1958.

[12] Tsichritzis, D., "The Equivalence Problem of Simple Programs", JACM 17,4,1970, 729-738.

[13] Hunt, H.B. III, D.J. Rosenkrantz, T.G. Szymanski, "On the equivalence, containment and covering problems for the regular and context free languages", (to appear JCSS).

[14] Meyer, A.R., and D. Ritchie, "Computational Complexity and Program Structure", IBM Research paper May 15, 1967. (RC 1817)

Acknowledgments

We would like to thank Sharon Gunkel and Helene Jacobowitz for typing the manuscript.